Aalto University
School of Science
Master's Programme in Computer, Communication and Information Sciences

Ilkka Oksanen

5G microservice monitoring with Linux kernel

Master's Thesis
Espoo, May 27, 2019

Supervisors: Senior University Lecturer Vesa Hirvisalo, Aalto University

Advisor: Samu Toimela M.Sc. (Tech.)

Aalto University
School of Science
Master's Programme in Computer, Communication and Information Sciences

ABSTRACT OF MASTER'S THESIS

Author:	Ilkka Oksanen		
Title:	5G microservice monitoring with Linux kernel		
Date:	May 27, 2019	**Pages:**	7 + 57
Major:	Computer Science	**Code:**	SCI3042
Supervisors:	Senior University Lecturer Vesa Hirvisalo		
Advisor:	Samu Toimela M.Sc. (Tech.)		

Software industry is adopting a scalable microservice architecture at increasing pace. At the advent of 5G, this introduces major changes for the architectures of telecommunication systems as well. The telecommunications software is moving towards virtualized solutions in form of virtual machines, and more recently, containers. New monitoring solutions have emerged, to efficiently monitor microservices. These tools however can not provide as detailed view to internal functions of the software than what is possible with tools provided by an operating system.

Unfortunately, operating system level tracing tools are decreasingly available for the developers or system administrators. This is due to the fact that the virtualized cloud environment, working as a base for microservices, abstracts away the access to the runtime environment of the services.

This thesis researches viability of using Linux kernel tooling in microservice monitoring. The viability is explored with a proof of concept container providing access to some of the Linux kernels network monitoring features. The main focus is evaluating the performance overhead caused by the monitor.

It was found out that kernel tracing tools have a great potential for providing low overhead tracing data from microservices. However, the low overheads achieved in the networking context could not be reproduced reliably. In the benchmarks, the overhead of tracing rapidly increased as a function of the number of processors used. While the results cannot be generalized out of the networking context, the inconsistency in overhead makes Linux kernel monitoring tools less than ideal applications for a containerized microservice.

Keywords:	eBPF, Linux, kernel, microservice, monitoring, 5G, container
Language:	English

Aalto-yliopisto
Perustieteiden korkeakoulu
Tieto-, tietoliikenne- ja informaatiotekniikan maisteriohjelma

DIPLOMITYÖN TIIVISTELMÄ

Tekijä:	Ilkka Oksanen		
Työn nimi:	5G mikropalveluiden valvonta Linux kernelin avulla		
Päiväys:	27. Toukokuuta 2019	**Sivumäärä:**	7 + 57
Pääaine:	Tietotekniikka	**Koodi:**	SCI3042
Valvojat:	Vanhempi yliopistonlehtori Vesa Hirvisalo		
Ohjaaja:	Diplomi-insinööri Samu Toimela		

Ohjelmistoala on yhä suuremmassa määrin siirtymässä skaalautuvien mikropalveluiden käyttöön. 5G:n saapuessa myös tietoliikennejärjestelmien arkkitehtuureissa nähdään suuria muutoksia.Tietoliikennejärjestelmät ovat muun ohjelmistoalan mukana siirtymässä virtualisoituihin ratkaisuihin, kuten virtuaalikoneisiin ja viimeisimpänä kontteihin. Uuden arkkitehtuurin myötä palveluiden valvontaan on syntynyt mikropalveluihin erikoistuneita työkaluja. Nämä työkalut eivät kuitenkaan pysty kilpailemaan käyttöjärjestelmän tarjoamien työkalujen kanssa valvonnan yksityiskohtaisuudessa.

Valitettavasti käyttöjärjestelmätason valvontatyökalut ovat arkkitehtuurimuutoksen takia harvemmin ohjelmistokehittäjien ja ylläpitäjien ulottuvilla. Suuri syy tähän on se, että mikropalveluarkkitehtuurin myötä palvelut on virtualisoitu pilveen. Tällöin pääsyä palvelun suoritusympäristöön ei usein ole.

Tässä työssä tutkitaan, onko Linux-ytimen valvontatyökalujen hyödyntäminen mikropalveluiden valvonnassa kannattavaa. Kannattavuutta tutkitaan kontissa ajettavalla monitoriprototyypillä, joka tarjoaa pääsyn osaan Linux-ytimen verkonvalvonta-ominaisuuksista. Tutkimuksen pääpaino on selvittää monitorin vaikututus ajossa olevan järjestelmän suorituskykyyn.

Tutkimuksessa selvisi, että Linux-ytimen valvontatyökaluilla on optimitilanteessa mahdollista kerätä mikropalveluiden tilaan liittyvää valvontadataa ilman suurta vaikutusta suorituskykyyn. Epäsuotuisassa tilanteessa valvonnan vaikutus nousi kuitenkin merkittävästi. Verkkovalvonnan suhteellisen vaikutuksen havaittiin kasvavan laskentakuormaan käytettyjen prosessorien määrän funktiona.

Tuloksia verkkovalvonnasta ei voi suoraan yleistää verkkovalvontakontekstin ulkopuolelle. Valvonnan vaikutuksen kasvun vahva riippuvuus käytetyn isäntäkoneen ominaisuuksista kuitenkin tekee Linux-ytimen valvontatyökaluista epäideaalin ratkaisun mikropalveluiden valvontaan.

Asiasanat:	eBPF, Linux, kernel, mikropalvelu, valvonta, 5G, kontti
Kieli:	Englanti

Acknowledgements

I would like to thank my supervisor, Vesa Hirvisalo, for the valuable feedback and the supervision.

I would like to thank my advisor Samu Toimela for the interest in my topic and valuable feedback along the writing process.

I would like to thank Heikki Laaksonen for always pushing me towards the (even more) bleeding edge.

I would like to thank Markku Niiranen for giving me the opportunity to work with this topic.

Last but not the least, I would like to thank my family and friends for without their invaluable support.

This thesis was conducted at Nokia Oyj in Espoo, Finland.

Espoo, May 27, 2019

Ilkka Oksanen

Abbreviations and Acronyms

5G	5th generation (mobile networks)
BBU	Baseband unit
BPF	Berkeley Packet Filter
BCC	BPF compiler collection
cBPF	Classic Berkeley Packet Filter
CNI	Container network interface
CPU	Central processing unit
eBPF	Extended Berkeley Packet Filter
GPL	GNU general public licence
HTB	Hierarchical Token Bucket
HTTP	Hyper text transfer protocol
IP	Internet Protocol
IPC	Interprocess communication
IRQ	Interrupt request
JSON	JavaScript object notation
Kprobe	Kernel probe
MEC	Multi access edge computing
NAT	Network address translation
NIC	Network interface controller
PID	Process identifier
Qdisc	Queuing discipline
RAN	Radio access network
RRU	Remote radio unit
SPAN	Switched port analyzer
TC	Traffic control
TCP	Transmission control protocol
TLS	Transport layer security
Uprobe	User space probe
URL	Uniform resource locator
UTS	UNIX time-sharing system
XDP	eXpress data path

Contents

Chapter 1

Introduction

For any modern service, receiving accurate and timely information about how the software is functioning is crucial. From operation and administration perspective, the information about systems health is necessary to keep a service running.

The rapid adaptation of the microservice architecture and distributed cloud computing has introduced a major challenge to application monitoring and debugging. The cause for this is the vast increase in the scale and complexity of the services. The services which were once powered by a single database and a java monolith, may now run in thousands of containers and multiple data centers. The inevitable consequence of this transition is that actual deployment environment has been automated and abstracted away from the operations personnel.

New types of tools, such as ELK stack [20] and Prometheus [48] have emerged for collecting application specific logs and metrics from distributed systems. However, the lower level operating system assisted debugging techniques such as profiling and tracing have not kept up with the advance.

Monitoring and debugging tools have not all managed to keep up with the transition to microservices. Need for the information provided by these tools however has certainly not decreased.

At the same time, the monitoring toolkit of the Linux kernel is under constant development. Each new version offers more sophisticated tooling than the previous. The tools like Perf, eBPF and XDP are under active development and can be used to do very fine-grained instrumentation of running processes. While the tools offered by the Linux kernel are focused on gathering information in the scope a host rather than a service, information provided can still be useful in full scale microservice context.

1.1 Problem statement

5G cloud introduces a challenging monitoring environment, where the techniques traditionally used for monitoring the mobile networks will be inadequate for the task. This is due to the fact that the techniques are not applicable to the microservice architecture as such. In a similar fashion, the monitoring solutions used for microservices cannot completely fulfill the requirements of the 5G Cloud Radio Access Network.

The purpose of this thesis is to explore modern Linux kernel monitoring, tracing and debugging utilities and to evaluate their applicability in the context of 5G microservice architecture. In practice, this is done by implementing a containerized monitor microservice. The goal of this service is to provide access to some of the monitoring and debugging tools provided by Linux kernel, which has been lost in the architecture change.

1.2 Scope and methodology

In the context of microservice monitoring, scope of the implementation is restricted to network monitoring and selective mirroring of the monitored data. Network monitoring has been a relevant monitoring and debugging mechanism as long as networked computers have existed. In this thesis, specific target for monitoring is the HTTP protocol. This is due to the popularity of HTTP as a communication protocol between microservices.

The focus in the context of kernel tooling is eBPF. This is due to its flexibility and active development. eBPF enables doing the necessary packet analyzing and the mirroring decision solely in the kernel. This makes it attractive tool for selective packet mirroring performance-wise.

From the container perspective there are multiple focus points. The most important one is the required configuration for running the eBPF based monitor inside a container. The configuration challenges include researching the possible dependencies to the host kernel and the security challenges introduced by the monitor. The other important aspect of containers is that in addition to the monitor, the monitored applications are run in containers as well. Important part of the container context is to address the complications rising from the network namespacing.

The performance overhead of the implemented monitor is tested by running network traffic benchmarks with and without the monitor. The overall impact of the monitor is then measured from the differences observed in the monitored and non-monitored benchmark.

1.3 Contribution

The author implemented an eBPF based network monitor for microservices and benchmarked the performance overhead of the monitor. It was found out that kernel tracing and monitoring tools have a great potential for providing low overhead tracing data from microservices. At the best case, performance overhead for network monitoring benchmarks was around 5%. However the overhead of tracing rapidly increased as a function of the number of processors used. The performance impact at the worst case was nearly 80%. While the results cannot be generalized out of the networking context, the inconsistency in overhead puts Linux kernel monitoring tools at a serious disadvantage, when compared with readily available microservice monitoring tools.

1.4 Thesis outline

Chapter 2 discusses the background topics of the thesis. The emphasis on this chapter is providing insight on software monitoring, 5G and microservices. Chapter 3 discusses tools used in the thesis. The focus of the chapter is in containers and Linux kernel tools for tracing and monitoring. Chapter 4 covers the implementation details of the network monitor and the test setup constructed around it. The network monitor part discusses the program logic of the monitor and the technologies used in the implementation. The test setup part describes the tools, both software and hardware, used for benchmarking the implemented monitor. Chapter 5 presents the results of the benchmarks and configuration. In Chapter 6 the results and their implications are discussed. Chapter 7 summarizes the implementation, obtained results and the discussion.

Chapter 2

Background

This chapter provides basic information about the topics of this thesis. First the telecommunications environment and its relationship to the virtualization and microservices are discussed. After that, there is an introduction to microservice monitoring and network monitoring tools and concepts.

2.1 Telecommunications environment for monitoring

5G comes with unprecedented technical requirements for the infrastructure. According to a survey conducted by Agiwal et al. [1], average user is expected to download a terabyte per year by 2020 via mobile networks. These forecasts can be seen in the requirements for supporting up to thousand times higher data volumes[56]. Most of this increase is expected to be originating from multimedia streaming. The eight core requirements for the 5G are:[1]

- 1-10 gbps data rate
- 1ms round trip time
- high bandwidth per unit
- enormous number of connected devices
- 99.999% availability
- 100% coverage
- almost 90% reduction in energy usage.

- high battery life

Besides the data rate, the requirement for ultra-low latency demands architectural change for the mobile networks. The sub millisecond latencies set a strict upper bound for the distance. Just the time light spends travelling in an optic fiber sets the maximum distance to around 100km. When the inevitable processing delays and possible network congestion is considered, the actual maximum distance will be a fraction of the 100km [54]. The implication of this is that applications requiring this level of latencies cannot be provided by a big data center possibly multiple hundreds of kilometers away from the user.

The edge cloud is a concept, which aims to move the computing closer to the user. This is done by having cloudlets between the user and the centralized cloud environment. According to the Satyanarayanan, the author of the term [51]

> A Cloudlet is a trusted, resource-rich computer or cluster of computers that's well-connected to the Internet and available for use by nearby mobile devices.

The solution is therefore to migrate the latency sensitive applications from the centralized cloud environment to the cloudlets running at the edge of the cloud. In the context of the mobile cloud computing, this is referred to as multi access edge computing (MEC). MEC is promised to provide capabilities for providing services directly at radio access network (RAN) edge [26]. This way, services with ultra-low latency can be provided without ever passing the traffic to the core network. Additionally, the services running at RAN edge may take advantage the edge context, and provide context, such as location, aware services [56].

Besides the edge cloud, virtualizing the mobile network infrastructure is a major ongoing process. the systems which used to run on top of specialized hardware, are now being moved to general purpose hardware on top of a virtualized layer [50]. There is multiple motivations for the virtualization. Arguably one of the most important features offered by virtualization is the flexibility with resource allocation.

An example use case to show case the benefits of virtualization in the telecommunications infrastructure is the virtualization of base band units (BBU). In the pre 3G world the BBUs were integrated to the base stations (BS) along with the remote radio units (RRU). Transition towards 4G first separated the BBU from RRU in 3G and in 4G the BBUs were moved to a centralized BBU pool and service multiple RRUs. In the Cloud Radio Access Network architecture, the BBU pool is virtualized and can be run

with general purpose hardware. In this case, the benefits of virtualization enable moving the BBUs from the radio sites to a centralized location, which reduces the costs. Furthermore, the load for each cell can have large variation as a function of the time of the day. The virtualization enables scaling the BBUs up and down as a reaction to the load, further increasing efficiency.

2.2 Service Monitoring

Software monitoring used to revolve around statically configured tools, such as Nagios, monitoring certain metrics. The base for these metrics is usually low-level usage metrics, such as memory, CPU and disk usage. Additionally, these metrics include application type specific polling of metrics to determine the health information. For example, in the web context this is polling is often done to some statically defined urls. The health of the system is then determined using some predetermined thresholds for the metrics. These methods are effective, given that the set of monitored applications remain static. With the adaptation of containers and microservice architecture, the once static resources changed to dynamic ones. Running instances of containerized services may be short lived by their nature and be rapidly moved from one host to another as a reaction to the events in the environment.

Because the statically configured tools do not handle the highly dynamic services well enough, new branch of monitoring tools specifically designed for dynamic configurations emerged. Examples of this kind of tools are the ELK stack [20] and Prometheus [48]. The work flow for these dynamic tools is follows. The first stage in the data generation is some unit included in the container itself which is responsible for collecting and emitting a metric. in ELK, these collectors are referred to as beats. These metrics are then parsed to uniform format and inserted to a centralized database. In ELK stack this parsing is done by Logstash and elastic search works as the database. The data is then visualized by a frontend, which in ELKs case is Kibana. The advantage of including the metric collectors in the containers is, that no configuration changes are required per newly launched container, as they automatically upload their metrics to the log collector service. Even though the configuration is significantly lighter than with the static tools, every type of service will still need a custom log emitter and log parser.

2.3 Network monitoring and data mirroring

Network monitoring has been popular tool as long as networked applications have existed. The network traffic analysis is widely used for example as a part of intrusion detection system, gathering analytic data or diagnosing behavior of a service for debugging purposes.

The network monitoring and especially traffic mirroring is always a trade-off between performance and the sophistication of the filtering applied. With no filtering, the data can be mirrored for example at router level with low overhead using technologies like Cisco SPAN [59]. On large systems, this kind of unfiltered mirroring however produces huge amount of data. Storing or moving all network traffic for centralized analysis may be unfeasible for being too expensive or even impossible.

The other extreme is capturing the data from an endpoint in user space using for example libpcap [57]. This approach allows for arbitrarily complex filtering logic bounded only by available computational resources and is a widely used especially for network related debugging. However, the filtering and processing overhead can often be undesirably high.

Multiple solutions have been proposed for handling the load [4] [17][42]. Common for all the approaches is the usage of libpcap to gather the (mostly unfiltered) data and then filter it and pass it further to usually centralized analysis. For handling the load itself, Zhao et al. [60] have proposed a solution based on submitting the data gathered via libpcap to a distributed message queue, which is able to process the large data volume. In a similar fashion, Laboshin et al. [38] propose mirroring the data to a computing cluster to be analyzed via (map-reduce) big data tools. Outsourcing the processing of network traffic to an external computing cluster has local overhead too. Any node in the network from which the traffic is captured, must process the packets at least to the extent of forwarding them to another interface. In practice however, the solutions using libpcap for capturing the packets need to pay the computing cost of copying each packet to the user space. In case of using for example a distributed message queue as the next step, encapsulation of the packets needs to be done as well. While the computing cost per packet might not be high, the cost starts to be significant, when the link speeds rise from less than gigabit per seconds to 10G or even 40G. Bonelli et al. [4] focus on solving the computational cost in the capture end. They propose a modified version of libpcap which is capable fanning out the packets to multiple cores. This does not reduce the processing time needed by the network capture but is nonetheless useful especially in case of high networking throughput and multiple available cores.

Network monitoring on the host systems is not the only way for capturing the relevant traffic between the microservices. The other solutions include for example forwarding the traffic through a proxy and instrumenting the monitored applications themselves for the purpose. As an example of a proxy based approach Erlacher et al. [21] proposed a TLS traffic sniffing proxy for monitoring encrypted network traffic. The caveat of this kind of proxy based approach is that the proxy needs to be configured for use for all applications and it introduces a single point of failure for the applications.

Examples of a solution which instrument the application themselves are Zipkin [62] and Jaeger [32]. The advantage of this solution is that much information regarding the context of the data can be included by the application and if needed, the data can be filtered without significant overhead on the fly. These tools also include sophisticated methods for tracing the requests trough a distributed system. The caveat of this approach is that each application needs to support the instrumentation. At least for the most popular http frameworks though, the effort for adding the instrumentation Zipkin for Jaeger is low, as the instrumentation is usually readily available via plugins.

Chapter 3

Tools

This chapter introduces tools used this thesis. The first part of the chapter is dedicated to introduction to the container technology. The second part discusses monitoring tools provided by the Linux kernel.

3.1 Containers

Containers are an operating system level virtualization technology. Instead of virtualizing whole operating system, containers leverage the isolation features of the an operating system itself to provide the containerized service an isolated view of the system. When compared to the virtual machines the containers have the advantages of being more lightweight, allowing for much faster startup times and greater density per host [50].

In the Linux, the isolation required by containers is provided by namespaces, cgroups and seccomp policies. Cgroups allow grouping processes and limit the resources available for the groups [8]. Seccomp on the other hand limits system calls available for the process [52]. The isolation can be further enhanced by mandatory access control such as SELinux [53] or AppArmor [2]. Linux namespaces allow the processes running in a namespace to have their own isolated view of a global system resource. Currently there is 7 available namespace types [39].

- **CGroup** namespaces provides isolation for the cgroup hierarchy. Every namespace has their separate root for the hierarchy.
- **IPC** namespace isolates inter process communication channels provided by the kernel. For example, POSIX message queues.
- **Network** namespace provides isolated view of the kernel network stack,

including for example interfaces and iptables rules. The network namespace is discussed further in the section 3.1.1.

- **Mount** namespace is used to isolate the mount points visible to a process [41]
- **PID** namespace isolates the view to the Linux kernels process table. This means that the namespaced processes will have no information about the processes outside their own namespace. A process not in the host namespace will have different PIDs depending on the namespace. The first process in a PID namespace will always the init process (pid 1) for the namespace. This init process has some special properties. For example, it will receive no signals from its namespace for which it has not set a handler. This means that it cannot be killed by its peers in the same namespace, regardless of their privileges, unless explicit signal handler is provided. Another special property of the init process is that when terminated, all the remaining processes in the namespace are terminated with sigkill. [46]
- **User** namespace allows the process to have different user and group id depending on the namespace. Processes may also run with root privileges which are effective only inside that user namespace. [47]
- **UTS** namespace provides isolation for the system hostname and NIS domain name [39].

The guarantees about isolation and self-containment provided by containers would not be very useful without easy way to manage the containers on a large scale. Container orchestrators such as Kubernetes[36] and Docker swarm[15] are designed to solve this exact problem. In addition to the basic management operations, such as launching and updating the containers, the orchestrators provide features for centralized management of a cluster of nodes running containers. Orchestrators in general have following key capabilities[34].

- cluster state management and scheduling
- providing high availability and fault tolerance
- ensuring security
- simplifying networking
- enabling service discovery

- making continuous deployment possible
- providing monitoring and governance.

From monitoring point of view orchestrators may both simplify and complicate matters. The orchestrators often provide access to basic container metrics. However, for example complex networking solutions may complicate network monitor based solutions.

3.1.1 Container networking

There is fundamental need for isolating the networking functions of the containers. One of the most important reasons is avoiding resource collisions, for example with ports. Another reason for isolating the networking resources of containers is the exposure. Another aspect is the security. If the services are exposed only to the applications that need them, the exposure of any vulnerability in the services is greatly reduced.

At the core of isolating the networking resources of the containers are the network namespaces provided by the Linux kernel. Each of the network namespaces have their own isolated view of the network stack, including for example the loopback interface and iptables rules. [43] How the network namespaces are utilized for the containers however depends on the used container orchestrator and configuration. As most of the used methods are in one way or another implemented in the docker ecosystem, it is used as example.

The most common way of isolating the container is using a veth-pair. In docker, this is referred to as the bridged networking mode. In this mode, each container is running on its own network namespace. A pair of connected virtual Ethernet interfaces is created, one in the containers network namespace and the other in the hosts namespace. The host side of the pair is connected to a bridge which allows the containers to communicate with each other and the outside world. In docker this bridge is named docker0.

By default, Docker creates a network namespace for each container. This is however not the only possible approach. The network namespace can as well be shared with multiple containers. In Docker, this is referred to as the Container mode. For example, Kubernetes uses this approach to share the network namespace within the containers belonging to same pod. [24]

Yet another approach is using macvlan. Macvlan allows adding multiple mac addresses to a single physical interface. Vlan is then used to distinguish between the subinterfaces. These subinterfaces may then be assigned to different network namespaces, allowing the required isolation for the containers. the macvlan approach is praised for its performance when compared to the

veth-pair approach [12][61]. The caveat is that it is not supported by most of the cloud providers. For example, Docker documentation suggests, that this approach should only be used for legacy applications, which expect access to the physical network interface. Furthermore, for example Kubernetes only supports this mode through third party CNI plugins such as multus-cni [37].

The networking model in Kubernetes is all in all more loosely specified than in Docker. The networking is based on following fundamentals. [24]

- Containers can communicate with all other containers without NAT.
- Nodes can communicate with all containers (and vice versa) without NAT.
- The IP a container sees itself is the same IP as others see it.

Kubernetes itself does not itself include the state-of-the-art tools for fulfilling these requirements, but instead leaves many of the features up to external implementation [24]. These implementations vary from from simple overlay networks such as flannel [22] to complex SDNs such as OpenVSwitch (OVS)[44] [37].

3.2 Kernel Monitoring

Linux kernel provides a wide variety of tools for obtaining information about running processes and the kernel state. This section introduces the basic concepts and the methods used for monitoring programs with the Linux kernel. First, the information sources available and provided by kernel are discussed. after that tools utilizing these sources are introduced. The eBPF and Traffic control are discussed in more details due to their relevance for the implementation part of this thesis.

3.2.1 Static Tracepoints

Static tracepoints are tool for the software developers to leave predetermined hookable points to the code. Instrumenting important parts of the code can significantly help tracing the software.

Many popular programs have a number of static tracepoints built in to their binaries [58]. The most notable example of this is the Linux kernel itself, which contains over 1500 tracepoints. the traceable kernel events can be listed with command:

```
cat /sys/kernel/debug/tracing/available_events
```

Some programs have the static tracepoints but omit them from production binaries. Usually in these cases the program needs to be recompiled with the tracepoints before they can be used [58].

On the low level, the tracepoints are implemented as a no-op check followed by a conditional jump. When the tracepoint is disabled, the execution program continues to the instruction after the jump. Enabling of the tracepoint is done by replacing the no-op check in the code. When the check is false, the program flow jumps to the tracepoint handling. The overhead of a disabled tracepoint consists of executing the no-op check. This is in practice considered negligible [3].

The status and the handlers for a tracepoint can be controlled via multiple different interfaces. Tracepoints can be controlled via debugfs interface provided by Linux kernel. More common approach however is using tracing toolkits such as SystemTap [55], LTTng [40] or BCC [5].

3.2.2 Dynamic tracing

Often the static tracepoints may prove insufficient for debugging purpose. To that extent there is a need for adding dynamic instrumentation in places where static tracepoints are missing. The Linux kernel provides this functionality in the form of Kprobes and Uprobes. Kprobes are method introduced in Linux kernel to add dynamic breakpoints to nearly any instruction residing in the kernel code. [33]. Uprobes on the other hand are used to instrument user space code.

The underlying principle behind the Kprobes is the usage of traps. When installing the probe, the trapped instruction is copied and replaced with instruction sets breakpoint instruction. The instruction is *int*3 in x86 architecture. When this breakpoint instruction is hit, kernels breakpoint handler is invoked and the state of currently running process is saved. Control is then given to the user registered probe function. After the user defined handler for the probe has returned, the state of interrupted program is restored, and the replaced instruction is executed. [33][23]

When compared to the other tracing methods executing the Kprobe by using the trap instruction is considered slow. [33][23] Mainly due to this considerable overhead, the later versions of the Linux kernel by using jump instructions instead of the break point instruction to direct the execution flow to a custom handler. This is referred as "jump optimization" or "trampoline" depending on the document. The jump optimization works by first jumping to a detour buffer where the execution context is saved to the stack. From the detour buffer, the execution is passed to the handler function registered to the probe and after the handler, back to the detour buffer.The detour

buffer then contains the instructions to restore the previously saved stack and jump back to the interrupted program. The detour buffer is constructed when the Kprobe is attached. [33][23]

The jump-based approach yields significantly lower overhead when compared to the trap based approach. There are however limitations for its usage. For example, the replaced instruction must not include call instruction or be a target of a jump. The kernel handles the optimizations and falls back to the trap-based approach if a jump cannot be used.

Uprobes are used in a similar way as the Kprobes. The underlying working principle is also the same. The biggest difference in usage between Kprobes and Uprobes is that user has to manually enter the address of the function traced when using the Uprobes [16]. With Kprobes, the functions may be referred with their name.

3.2.3 Extended Berkeley Packet Filter (eBPF)

Extended Berkeley Packet Filter (eBPF) is a light weight virtual machine-like construct in the Linux kernel. The eBPF allows bytecode attached from a process running in the user space to be hooked to certain events occurring in the kernel. [9] The important factor is that the bytecode itself is run in the kernel space, as this eliminates the need for passing data and control to some user mode function for every traced event. This avoids a significant performance penalty. Especially so if the monitored events are occurring frequently.

The eBPF programs attached to kernel events may modify handling of these event inside the kernel. The programs have access to the context of the event via kernel supplied pointer and furthermore, the return value on a eBPF program determines an context dependent reaction from the Linux kernel to the event. For example, in a case of a socket filter program, the reaction determines whether to forward the packet to the user space or not.

The work flow for an eBPF program is illustrated in the Figure 3.1. First the eBPF program is written in a subset of C language. Then Clang and LLVM can be used to compile the program to eBPF bytecode. This compilation requires Linux kernel sources to be present in the system. The compiled byte code can be attached to some available hook point in the Linux kernel. Before the eBPF program is loaded into the kernel it is verified by static verifier. if the verification passes, the program is then loaded to the kernel. Often the steps of loading the program to the kernel are done by a separate user space program which is responsible for extracting the data from the loaded program as well. Some types of BPF program however can be attached without an user space counterpart. An example of such would

be a traffic control filter, which can be loaded directly via the user space traffic control tool "tc".

3.2.3.1 BCC

BCC is a toolkit to streamline the usage of eBPF. It is an open source project maintained by IOVisor. It provides a python frontend for loading, attaching and interacting with eBPF programs [5]. The work flow of using BCC to manage eBPF program is different to what was described above. With BCC, there is always a user space program controlling the eBPF program. The big difference between BCC and the traditional work flow is that BCC compiles the eBPF programs on the fly from the user space program.

3.2.3.2 eBPF Virtual Machine

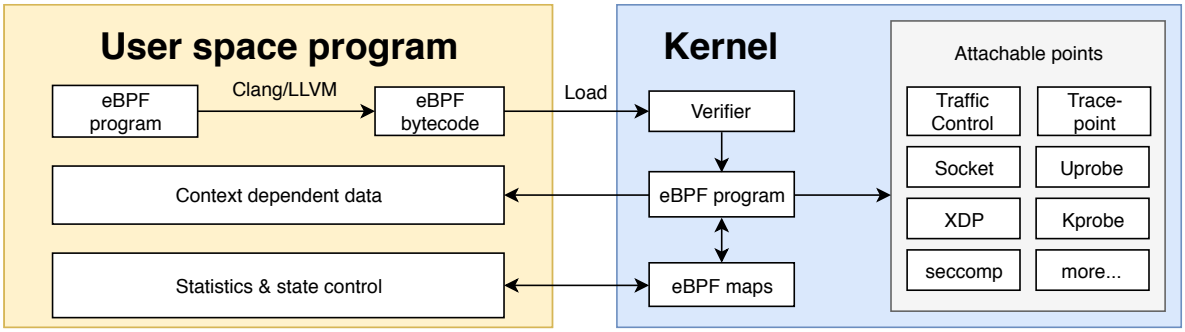

Figure 3.1: eBPF program view. Based on [18]

The eBPF virtual machine includes 11 64bit registers and 512 bytes of stack space. Furthermore, the size of a program is limited to 4096 instructions. The instruction set is by design fairly restricted and designed to be easily mappable to many modern instruction sets, such as x86 or ARM. In the optimal case eBPF instructions can mostly be mapped to a single instruction in the underlying architecture.

3.2.3.3 Verification

Before being attached to an event, every eBPF program is verified by an in-kernel verifier. The verifier guarantees that a the eBPF program will terminate and that it may not crash. These guarantees are crucial in allowing user space provided code to be run in the kernel as non-terminating program run in kernel space would very likely result in a unresponsive system and a crashing program run in kernel mode will result in a kernel panic.

The verifier is able to provide such guarantees due to the limitation introduced to the BPF programs. The most important limitation is that no loops are allowed. This implies that the BPF environment is not Turing complete.

3.2.3.4 Helper functions

Every eBPF program type has their own subset of helper functions available for use. Some of the functions are very general purpose, such as getting a processor time, and can be used with nearly any program type. Others are heavily context dependent. For example, a helper which reads bytes from a socket buffer only makes sense in a program with networking context.

3.2.4 Licensing

The license used by the eBPF program affects what the program can do. This is a consequence of the fact that some of the helper functions are GPL licensed and can thus only be used if the program itself is GPL compatible. The GPL licensed helpers are mostly related to the perf events provided by the Linux kernel. The perf related use cases can therefore be heavily restricted in a commercial use. The licensing of a program is indicated to the kernel verifier by adding equivalent of a c code line

```
char ____license[]
     __attribute__((section("license"), used)) = "GPL";
```

to the eBPF program.

3.2.5 Traffic control

Data mirroring and the hook point for the eBPF program are both provided by the traffic control (TC). As described by Martin Brown [6]:

> Traffic control is the name given to the sets of queuing systems and mechanisms by which packets are received and transmitted on a router. This includes deciding which (and whether) packets to accept at what rate on the input of an interface and determining which packets to transmit in what order at what rate on the output of an interface."

In the network processing order of the Linux kernel, the traffic control resides outside of the network stack. The ingress packets enter the network stack only after passing through the traffic control. In a similar fashion,

the egress packets are handled by the traffic control only after leaving the network stack.

The fundamental component of the traffic control is a queuing discipline, often referred to as qdisc. A responsibility of a qdisc is to handle enqueuing and dequeuing packets. An example of a simple qdisc would be just a FIFO (first-in first-out) queue. A queuing discipline may however use arbitrarily complex internal mechanics to decide whether to enqueue the packet at all and in which order to dequeue enqueued packets. Features of the more complex queuing disciplines for example enable policing or prioritizing packets matching certain classification.

Classes are another important component in the traffic control. Classes are containers for other TC components, which offer traffic shaping capabilities. A class itself always contains either a single qdisc or multiple classes. as the queue discipline contained by a class can itself be classful, construction of complex hierarchies of classes and queue disciplines is possible.

A third central component in the traffic control is filters. Filters can be attached to classful qdiscs and classes. The purpose of filter is to classify packets going through its attach point. For example, in a case where a class attached to a qdisc has multiple child classes, a filter is needed determine to which child a packet should be assigned. In addition to classifying packets, filters may take several other actions. These actions include dropping, marking or triggering a reclassification for a packet. A filter can also decide to take no action about a packet in which case it is passed to the next filter attached to same class (or qdisc).

There is distinct hook point for queuing discipline for egress and ingress traffic of each interface. For egress traffic the top level qdisc is called root. The root qdisc defaults to qdisc called fast_fifo but can be replaced with any available qdisc. [27] While any qdisc can be assigned as the root qdisc for egress, the same is not true for the ingress. The top level qdisc for ingress is just called ingress. The ingress qdisc cannot be replaced and no classes can be attached to it. Filters can however be attached. This limits the functionality of the ingress qdisc when compared to the egress ones.

Chapter 4

Kernel monitoring benchmark

This chapter introduces an eBPF based traffic monitor implemented for this thesis. In addition to the monitor, this chapter describes a test setup to benchmark the implemented monitor.

4.1 Traffic monitor

In order to determine the performance overhead of an eBPF based traffic monitor. A simple test monitor is benchmarked against several different types of test traffic. The purpose of these benchmarks is to find out the overhead of monitoring very high frequency events purely inside the Linux kernel. Monitoring network traffic is however a reasonable choice for benchmarkable use case due to the naturally high frequency of the networking events. Due to increasing amount of hook points for the eBPF programs in the Linux kernel and the flexibility of eBPF maps, the possible use cases of eBPF based monitoring are vastly larger than will be used in the benchmark. The traffic monitor itself will be an eBPF program, which searches through the traffic for HTTP status codes. When an error indicating status code is detected in the traffic, the corresponding TCP stream will be mirrored to a debug network interface. HTTP was chosen as monitored protocol due to its wide use as an unified communication protocol between microservices. The eBPF program itself is inserted to the Linux kernels traffic control subsystem as a filter. The packets dropped by the filter are then mirrored to the debug interface by traffic controls mirred-action.

The eBPF monitor, as well as the traffic generators, will be implemented as a docker containers. the containerized approach was chosen due to its popularity in microservice implementations.

4.1.1 Implementation details

The implemented eBPF monitor is composed of two different parts. The first part of the monitor is the eBPF program, which is attached to the kernel and is responsible for the actual monitoring. The second part is the user space program managing the traffic control configuration and attaching the eBPF program to the kernel.

The implementation was done using BCC tools. As usual in the BCC workflow, the user space program is done using python and the bindings provided by the BCC library. The first thing the program does is insert proper traffic control queue discipline to the monitored interface. This is done using pythons pyroute2 [49] library, which provides the Linux Iproute2 bindings for the python. At first the BPF program is loaded and compiled from a file.

```
from bcc import BPF
from pyroute2 import IPRoute

ip = IPRoute()

bpf = BPF(src_file="bpf_source.c", debug=0)
mirror = bpf.load_func("mirror", BPF.SCHED_CLS)
```

After the eBPF program is loaded a new dummy interface is created and enabled.

```
ip.link('add', ifname='dummy0', kind='dummy')

# lookup the indexes of the used interfaces
dummy = ip.link_lookup(ifname='dummy0')[0]
eth   = ip.link_lookup(ifname='eth0')[0]

ip.link('set', index=dummy, state='up')
```

After the interfaces are configured, the needed queue disciplines (qdisc) must be added to the interface we want to monitor. To monitor the incoming traffic, a special qdisc of type ingress is added to the interface. Adding the qdisc is required since a traffic control filter of any type requires a qdisc class to be able to be attached and by default the ingress traffic does not have one. Contrary to the ingress the egress traffic does have a default qdisc. the default however does not support adding filters, so it is replaced by a qdisc called Hierarchy Token Bucket (HTB) which has the support.

```
ip.tc('add', kind='ingress', index=eth, handle='ffff:')
ip.tc('add', kind='htb',     index=eth, handle='1:')
```

When the proper qdiscs are in place, it is possible to add the eBPF filters. The mirror action parameters shown below specify that the mirrored packets should appear as egress packets on the interface dummy. The eBPF program itself is passed to the traffic control via the file descriptor available through previously used eBPF load_func command.

```
# define the mirror action used in filters below
mirror_action = [{
    'kind': 'mirred',   'direction': 'egress',
    'action': 'mirror', 'ifindex': dummy
}]
# Add the ingress filter
ip.tc(
    'add-filter',     kind='bpf',
    index=eth,        handle=':1',
    name=mirror.name, fd=mirror.fd,
    parent='ffff:',   action=mirror_action
)
# Add the egress filter
ip.tc(
    'add-filter',     kind='bpf',
    index=eth,        handle=':1',
    name=mirror.name, fd=mirror.fd,
    parent='1:',      action=mirror_action
)
```

The filtering logic of the eBPF program is illustrated in the Figure 4.1. The program first scans through the Ethernet, IP and TCP headers and in each case, quits early if the protocol is wrong. in the case the packet is indeed a TCP packet, its source and destination IP address and port are looked up in the map of traced connections. If a match is found from the map the packet is immediately destined to be mirrored. Otherwise the beginning of the packet is searched for the beginning of a HTTP response and if found, the status code is read as well. For testing purposes, the status code is 500, indicating a server error, was selected as a trigger for mirroring the connection. when trigger is found, the connection is added to the map of traced connections and the packet is classified for mirroring. With any other status code, the packet is just passed through normally.

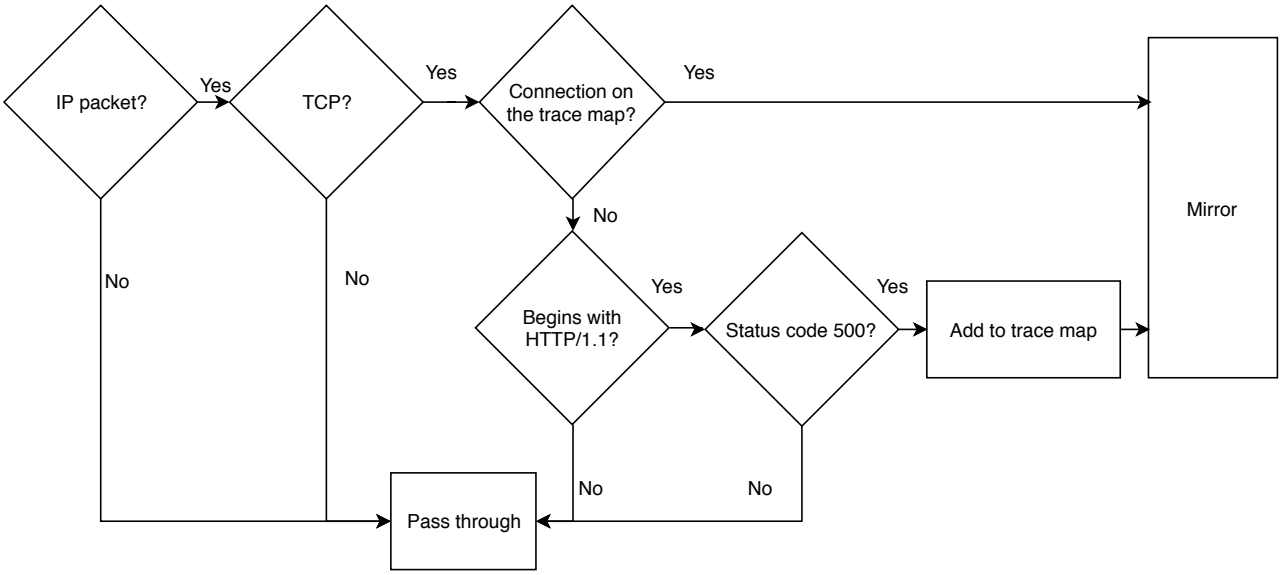

Figure 4.1: Tracer classification logic

The connection map is implemented using a least recently used hash map. This means that least recently active connections will be dropped in case the the maximum number of traced connections is exceeded. In a production grade monitor, more sophisticated solution would be recommended for removing inactive connections. However for testing purposes, the least recently used map provided a simple solution. The key used in the hash map to identify connection is following C language struct.

```
struct ConnectionInfo {
        uint32_t src_ip;
        uint32_t dst_ip;
        uint16_t src_port;
        uint16_t dst_port;
};
```

The value of inserted into the map is not relevant to the function of the monitor. In a more advanced version however it could be used as a timestamp to timeout the tracing of a connection if no further trigger event have not occurred.

the eBPF program may return one of two different return values depending on the desired outcome. To just pass the packet through, the program returns TC_ACT_OK. For mirroring the relevant return value is TC_ACT_SHOT which is in general used to drop the packets. The mirroring is then implemented as action for the packets dropped by the eBPF filter,

which results in the desired behaviour. All possible return values are defined in the Linux kernel header "pkt_cls.h" [45].

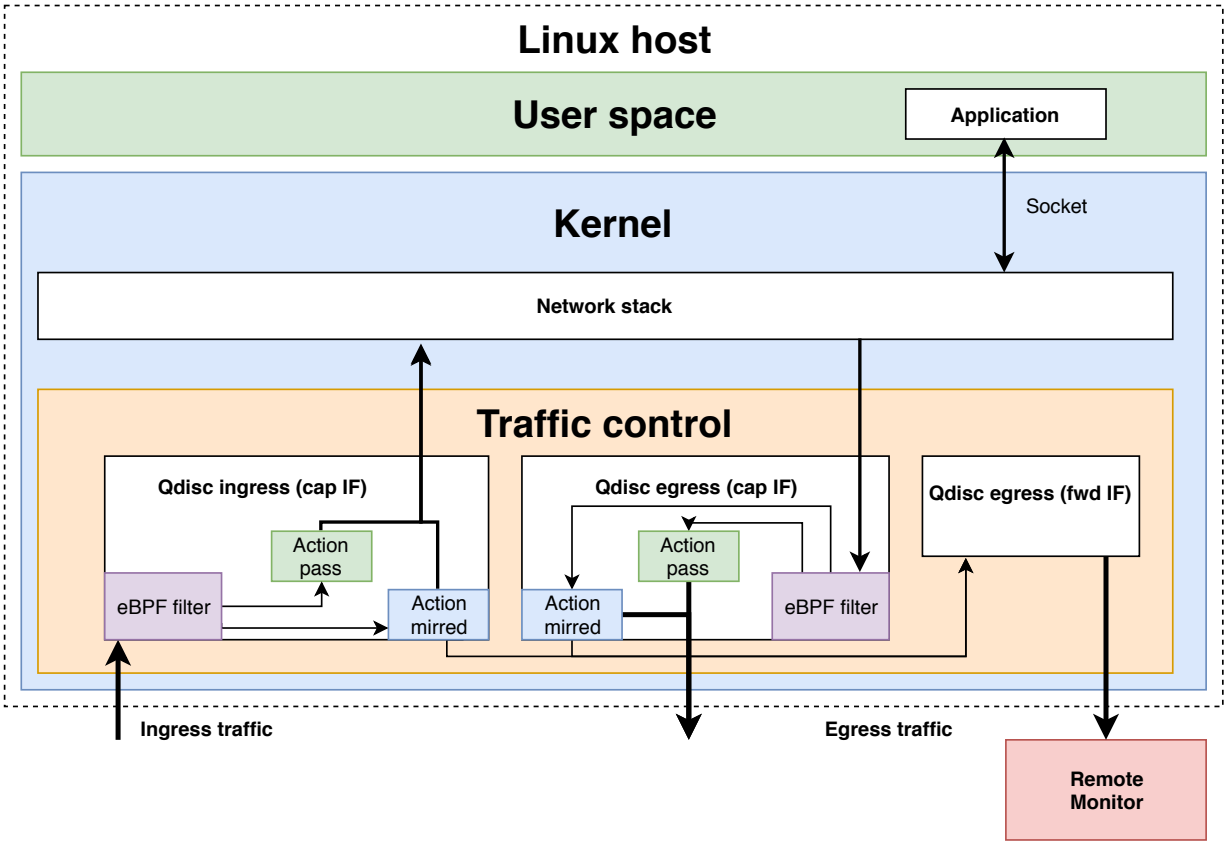

Figure 4.2: Networking view of the tracer architecture

4.1.2 Containerization and host requirements

The monitor was built to be run in a Docker image. This introduces challenges for running the monitor. The reason for this is that by default, Docker restricts some of the features of Linux kernel which are required by the eBPF monitor. The additional flags and configurations to run the monitor were implemented to be equal to following docker command.

```
docker run \
    --network host \
    --cap-add NET_ADMIN \
    --cap-add SYS_ADMIN \
    -v /lib/modules:/lib/modules \
```

```
-v /usr/src/:/usr/src \
monitor_image_name
```

The option '–network host' is needed due to the network namespacing. By default, each container will be in their own network namespace which is undesirable for the monitor as it is unable to see the traffic necessary for its function. the line in question assigns the container directly to the host network namespace.

The '–cap-add' flags give the container the capabilities CAP_NET_ADMIN and CAP _SYS_ADMIN. [7] The CAP_NET_ADMIN is required by the monitoring to be able to insert the eBPF program to the traffic control. The CAP_SYS_ADMIN is instead required for the monitor be allowed to access the required eBPF maps for storing the connection information.

Compiling eBPF programs requires the sources of the kernel version running on the host system. The sources are therefore mounted from the host system with "-v" flags. /usr/src directory is the actual location of the kernel sources. /lib/modules directory has to be mounted to the container as well due to there being a symbolic link to a subfolder in /usr/src which the BCC tools is using to access the headers.

4.1.3 Limitations

The most important limitation of the implemented monitor is that only the start of each packet is scanned for the start of an HTTP response. Furthermore, it is assumed that the first 10 bytes of the headers are contained in the same packet. Neither of these conditions are necessarily true. As TCP is a purely stream oriented protocol. A client could send the header very slowly, even one byte at time. This would mean that the monitor, expecting at least 10 bytes, would never correctly parse a such request. In addition, HTTP requests may be pipelined by a sender which means that a start of an HTTP header may be in a middle of a TCP packet after a previous request. In these cases, the eBPF monitor would simply miss the trigger for mirroring.

In practice it was found out that even with request pipelining most of the first bytes of the headers did indeed end up being at the start of a TCP packet. This is most likely due to each request being sent with separate system call allowing the delay between the calls to cause a flush in the underlying buffer. The case of really slow sending of TCP packets where the first 10 bytes would not end up in the same packet is extremely unlikely to occur outside of a purposely adversarial scenario.

Further limitation is that the implemented monitor supports only the 1.0 and 1.1 versions of HTTP and IP version 4. IP version 6 support would be

a trivial addition but was deemed unnecessary for the testing purposes.

4.1.4 Metrofunnel, Reference monitor

Metrofunnel is a Restful request monitor based on capturing the HTTP traffic with libpcap [28]. Metrofunnel was chosen as an alternative monitor for the benchmark due to the fact that, its use case greatly resembles the implemented eBPF monitor. The main difference in the use is that instead of mirroring the network data, Metrofunnel keeps track of alive requests and measures the delay and the result. [11] The purpose of using alternative monitor is to compare eBPF results with a monitor for which there exists results in the literature.

4.2 Test setup

The implemented monitor is benchmarked with two different traffic patterns and two different host configurations. The first of the two is the Iperf3 bandwidth benchmarking tool. The purpose of using Iperf3 is to flood an interface with as much traffic as the interface can take and measure the difference in obtained bandwidth in a case where the monitor is present and in a case where it is not. Even though the mirroring mechanism is not tested at all by this load, the test with Iperf3 will show the impact of the monitors presence when a maximum number of packets is put through the monitored network interface.

Another used traffic pattern is a flood of small HTTP requests. This traffic pattern is generated by using two different tools. Nginx is used as the HTTP server due to its high performance. Furthermore, it is an industry standard for example as a HTTP proxy. h2load utility from HTTP2 library Nghttp2 is used as a HTTP client for the test. As with the Iperf3, the throughput of the traffic is measured with and without the monitor. This time however, varying amount of the traffic is mirrored through another interface.

Both traffic patterns will be tested in two different environments. The first environment is a single host server which runs both the traffic generators and the monitor. As the traffic goes through the loopback interface. It is to be expected that the setup is solely bound by processor or possibly memory speed. The scenario is not unrealistic because in a microservice architecture, there often is multiple services running on a single host which is part of a larger cluster.

The other environment is a multi-host environment. The environment contains two hosts, one of which is running the traffic generator client and the other will be running the traffic generator server. Both hosts are running their own instance of the monitor. Results obtained for the monitoring overhead in this environment should more closely match the overhead in a real microservice environment than with the single host.

4.2.1 Iperf3

Iperf3 [30] is a bandwidth measurement tool used in this research to measure TCP throughput. With the Iperf3 based test load, both the client and the server are running Iperf3. In both of the cases, a Iperf3 was ran on a Debian 9 based container provided at Dockerhub by name networkstatic/iperf3 [29]. The Iperf version used was 3.0.7. The test setup the server was run without any notable changes to configuration with following options.

```
iperf3 -s -B 0.0.0.0
```

The client in turn was started with following command:

```
iperf3  -i 1 -t 60 -J -Z -O 10 -c <ip address>
```

Option	Explanation
-i 1	Aggregate the statistics in one second intervals
-J	JSON output
-T 60	Duration in seconds
-Z	Zero copy mode for sending the data.
-O 10	Omit the first 10 seconds of the test
-c <ip address>	Client mode. Connect to the specified address

Table 4.1: Used Iperf3 client options [31]

The meaning of the options used above are explained in the table 4.1. The reason for using the zero copy mode is to maximize the throughput of the load generator and thus highlight overhead imposed by the monitor.

4.2.2 Nghttp2 and Nginx

In addition to the traffic generated with Iperf3, a combination of Nghttp2 and Nginx was used to generate HTTP traffic to test the monitor with load it is intended to parse. On the server side, Nginx was configured to respond to two different urls with following configuration.

```
location /foo {
    return 200 'foofoofoofoofoofoo';
}
location /bar {
    return 500 'bar';
}
```

The functionality of the two urls differs only on the response code sent and the length of the payload. One sends a 200 response implying a successful response and the other 500 indicating an internal server error. To maximize the performance, the payloads are short and are returned directly instead of reading from a file. The longer payload for the 200 response is to compensate the slightly longer response header in the error response. Namely in the first header line there is "Internal Server Error" instead of "OK". With the payload shown in the configuration above, the TCP payload length is same regardless of the queried url. In addition to the above urls, Nginx configuration was altered with following settings: Amount of worker connections was set to 8192, the number of worker processes was set to 2 and the number of keep alive requests was set to 10000.

On the client side, h2load tool from Nghttp2 library was used to generate http requests to the Nginx.

```
h2load -D 60 -c 10, -m 1 -N 15 -T 15 --h1 <URL>
```

Option	Explanation
-D 60	Duration in seconds
-c 10	The number of clients to use
-m 1	the maximum number of pipelined requests
-N 15	Active connection timeout in seconds
-T 15	Connection inactivity timeout in seconds
- -h1	Use HTTP version 1.1
<URL>	The URL to send the requests to

Table 4.2: Used h2load parameters

An example of a single generated HTTP request used in the benchmarks has 78 bytes TCP payload and is shown below.

```
GET /bar HTTP/1.1
Host: 127.0.0.1:8888
user-agent: h2load nghttp2/1.34.0
```

In turn a single response has a 183 bytes TCP payload.

```
HTTP/1.1 500 Internal Server Error
Server: nginx/1.15.7
Date: Mon, 17 Dec 2018 07:32:48 GMT
Content-Type: application/octet-stream
Content-Length: 3
Connection: keep-alive

bar
```

4.2.3 Single host configuration

Memory	8Gb DDR3 1600MT/s SODIMM
CPU Model	Intel Core i7-3740QM 2.70GHz
CPUs	8
Threads per core	2
L1i cache	32K
L1d cache	32K
L2 cache	256K
L3 cache	6144K

Table 4.3: Hardware

In the single host configurations all of the containers included in the benchmark were accommodated in a single computer. The network traffic in a single host configuration is going solely through the loopback interface. While the interaction of the monitor with container networking is a relevant topic in this research, the benchmarks aim for the greatest stress to the monitor. The container networking is thus omitted from the benchmarking. The container networking considerations to the monitor are detailed in the section 3.1.1.

The the single host configuration was run with Fedora 29 operating system running Linux kernel version 4.18.16-300. The hardware specification is described in detail in the table 4.3.

To reduce jitter in the results, intel_pstate was disabled. intel_pstate is a Linux kernel driver which controls the CPU frequency. The driver adapts the used frequency on the fly based on the load. The changes in frequency in turn causes jitter to the benchmarks. The disabling was done by adding

the line "intel_pstate=disable" to a file "/etc/default/grub". After that the grub configuration had to be updated and the machine rebooted.

Disabling the intel_pstate causes linux to use acpi-cpufreq driver to control which in the experiments was configured to use static 2.7GHz frequency.

4.2.4 Multi-host configuration

The Multi-host setup is built on two servers connected via 10G fiber connection. The hardware specification is described in the table 4.4. As with the single host configuration, the operating system running on the servers was Fedora 29. The running Linux kernel on the other hand was 4.20.16-200.fc29.x86_64, which is a few versions newer than the one used the single host configuration.

To simulate a busy containerized environment, the amount of running benchmarking instances were scaled up for the HTTP benchmark. Instead of the 5 h2load instances running on a single core, 48 instances were used spanning to all of the 48 cores of the server. The Nginx was scaled up respectively to a total of 48 worker processes. The total amount of monitors running was also raised to two. One for each of the host servers. This reflects the single host test in the sense that each of the packets hits a monitor twice.

The test setup was modified so that instead of 5 runs for different portion of mirrored traffic, there was only 3. Therefore, the tests were run for 0%, 50% and 100% of data mirrored.

The monitor itself was modified to use cpu local lru-caches for connection storing instead the stock lru-caches. The reason for this modification is excluding the overhead of the synchronization of the maps from the results.

Memory	126 Gb DDR4 2133 MT/s DIMM
CPU Model	Intel(R) Xeon(R) CPU E5-2680 v3 @ 2.50GHz
CPUs	48
Threads per core	2
L1i cache	32K
L1d cache	32K
L2 cache	256K
L3 cache	30720K

Table 4.4: Multi host Hardware

Chapter 5

Evaluation

This chapter presents the results of benchmarks described in the previous chapter. First the results with single host configuration are presented and analyzed. Then a similar presentation is done for the multi-host configuration. Finally this chapter presents experimentation with the test configurations based on the results obtained.

5.1 Single Host results

The single host benchmarks were conducted in three different configurations. The first is the non-monitored reference run. In graphs and tables this is referred to as the "untraced" benchmark. The second is the benchmark with the eBPF monitor in place. This is referred to the "traced" benchmark. The third configuration is referred to as "TC only". In this configuration, the queuing disciplines required to attach the eBPF filter were installed. However, the eBPF filter itself was not.

5.1.1 TCP throughput

The impact of the monitor to the TCP throughput was measured using the Iperf3 tool. For one test run the tool was run two times for 60 seconds, once with the monitor and once without. To reduce noise the results were averaged with a total of 801 runs of the same test. The averaged data for the throughput is shown in the table 5.1. The benchmarks show roughly a 10% lower TCP throughput with the monitor than without. The monitor has three parts which may affect the throughput. The HTB queuing discipline attached to the egress, the bpf filter itself, and the user space monitor program. The impact of the user space program should be minimal as it

is sleeping in 1 second interval during the test. The overhead of using the queuing disciplines which support filters at all amount to a quarter of the penalty in the throughput. The other three quarters can be attributed to the eBPF filter itself.

	Untraced	**Qdiscs only**	**Traced**
Average throughput	28.9 GBit/s	28.2 GBit/s	25.9 GBit/s
Standard deviation	0.26 GBit/s	0.31 GBit/s	0.76 GBit/s
Percent of untraced	100%	97.7%	89.7%

Table 5.1: Throughput test summary

5.1.2 HTTP

The results for the benchmark with HTTP traffic payload yield information about the impact of the monitor to the HTTP-request throughput and delay. Due to high amount jitter in the results, the values presented in this section are medians of the corresponding metrics. The median is taken from a total of 801 runs. As an example, the mean throughput is therefore the median of the means calculated for each run.

The HTTP throughput is plotted as a function of the error request percentage in the Figure 5.1. The presumption for the results is that the throughput of non-monitored and qdisc only versions is not affected at all by the percentage of the error request. The presumption for the monitored version is that the throughput decreases as the percentage of the error request increases. The basis for this presumption is that mirroring of the request has non-zero overhead. That overhead should increase as the percentage of error request and therefore mirrored data increases.

As per the presumptions, the HTTP throughput of the monitored traffic load indeed dropped in seemingly linear fashion from 97% of the non-monitored throughput to 89%. The results from raw TCP throughput benchmark would suggest that the benchmark with only the queuing disciplines in place should result in decreased throughput when compared to the non-monitored benchmark. however, this was not observed.

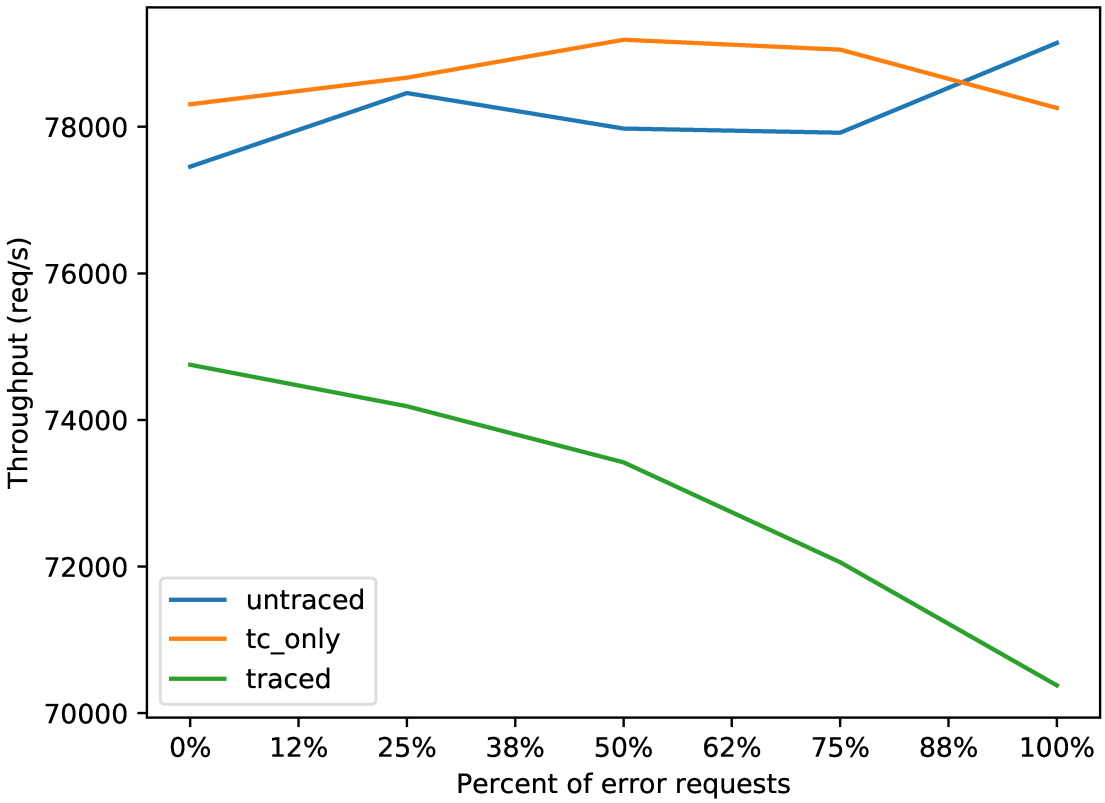

Figure 5.1: h2load throughput

The presumptions for the mean delay for the request were similar to the throughput. The only exception being that the delay was expected to increase instead of decreasing along with the error request percentage. Again, the reasoning for this suspected increase is the additional overhead of the mirroring. The mean delay as a function of request error percentage is plotted in the Figure 5.2. The results indicate that there is indeed a linear increase in delay from 5% to 13% when compared to the reference run. As with the throughput, there is not significant difference between the queue disciplines only and the reference run.

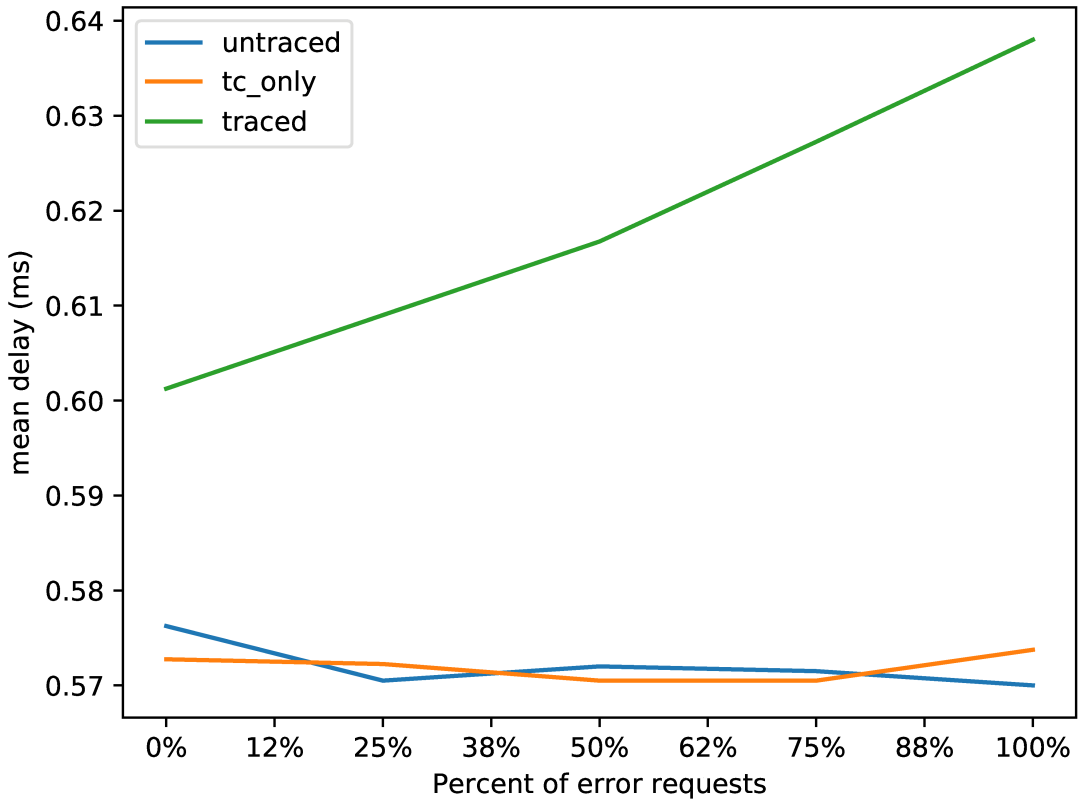

Figure 5.2: h2load mean request delay

The distribution of the request times is presented in Figure 5.3. The distribution histograms show that most of the requests (nearly 90%) are completed in a time between 128 and 256 µs regardless of used monitor or error ratio. More important than the distribution of the requests itself is the difference of the distributions between the monitored and the reference runs.

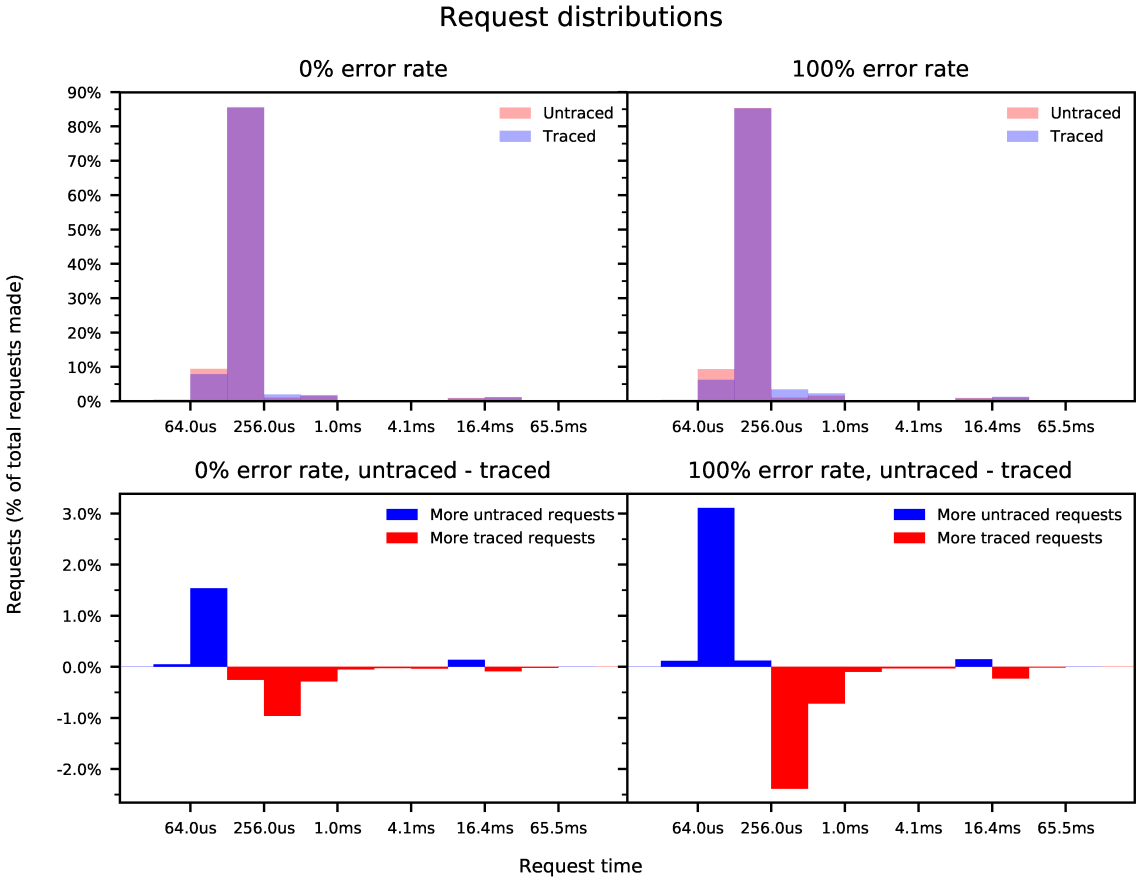

Figure 5.3: h2load request time distributions on log2 time scale. The second row shows the distribution difference between the monitored and non-monitored runs.

The difference in distribution is illustrated for with 0% and 100% error requests in the two bottom histograms of Figure 5.3. The histograms show that the difference in the request times is mostly composed by a shift of request times from the 64 - 128µs bucket to the buckets ranging from 256µs to 1ms. This shift in request times is amplified by the error request mirroring. For example, the difference in amount of requests in the 128 and 256 microseconds bucket increases from 1.5% of the total request amount to 3% when the the error request ratio is 100%.

Percentile	50	90	99	99.9	99.99	99.999	99.9999
untraced	0.154	0.200	18	32	46	57	66
tc_only	0.156	0.207	19	33	47	57	207
traced	0.164	0.226	19	33	48	57	207

Table 5.2: HTTP request time (ms) percentiles with 0% errors

The request percentiles in the tables 5.2 and 5.3 show that all meaningful differences in the request times happen before the 99.9th percentile regardless of the request error rate. The difference between the 99.9th percentiles with 100% error rate is at most 3%. The smaller percentiles are however affected by the monitoring and the error rate. Without any errors the median request time increases by 6% when the monitoring is introduced. With errors the same increase is 13%. The 99.9999 percentiles show a significant increase in delay for the tc_only and traced benchmarks. This seems to be connected to the HTB queuing discipline used. However, the exact reason is unknown.

Percentile	50	90	99	99.9	99.99	99.999	99.9999
untraced	0.154	0.200	18	32	47	57	65
tc_only	0.156	0.207	19	33	48	57	207
traced	0.174	0.244	20	33	47	57	206

Table 5.3: HTTP request time (ms) percentiles with 100% errors

5.2 Multi-host results

The Multi-host benchmarks were conducted in four different configurations. The "traced" and "untraced" configurations match their counterparts described in the single host configuration. The "funnel" represents configuration where the Metrofunnel was used for network monitoring instead of the eBPF monitor. In the fourth configuration the eBPF monitor used is identical to the one used in the traced configuration. However instead of mirroring, the filter will just pass the packets to the network stack without further action. This configuration is referred in tables and figures as "pass".

5.2.1 TCP throughput

The results for TCP throughput are presented in the table 5.4. The results presented are averaged between 70 runs of the Iperf3 benchmark. Due to the

fact that a single core used for generating the traffic is able to saturate the 10G NIC, the results are uneventful. When the computing is transformed from CPU bound to network bound, the overhead of monitoring vanishes completely.

	Average throughput	**Standard deviation**
Untraced	8.768 GBit/s	0.003 GBit/s
Traced	8.768 GBit/s	0.003 GBit/s

Table 5.4: Multi-host TCP throughput test summary

5.2.2 HTTP

The results of the HTTP throughput are presented in the figure 5.4.The HTTP throughput measured with the monitor with drops to 30% of the non-monitored reference without mirroring enabled. 100% mirroring causes the throughput drop even further to 20% of the reference. This is a drastic drop when compared to the 10% overhead measured in single host benchmark.

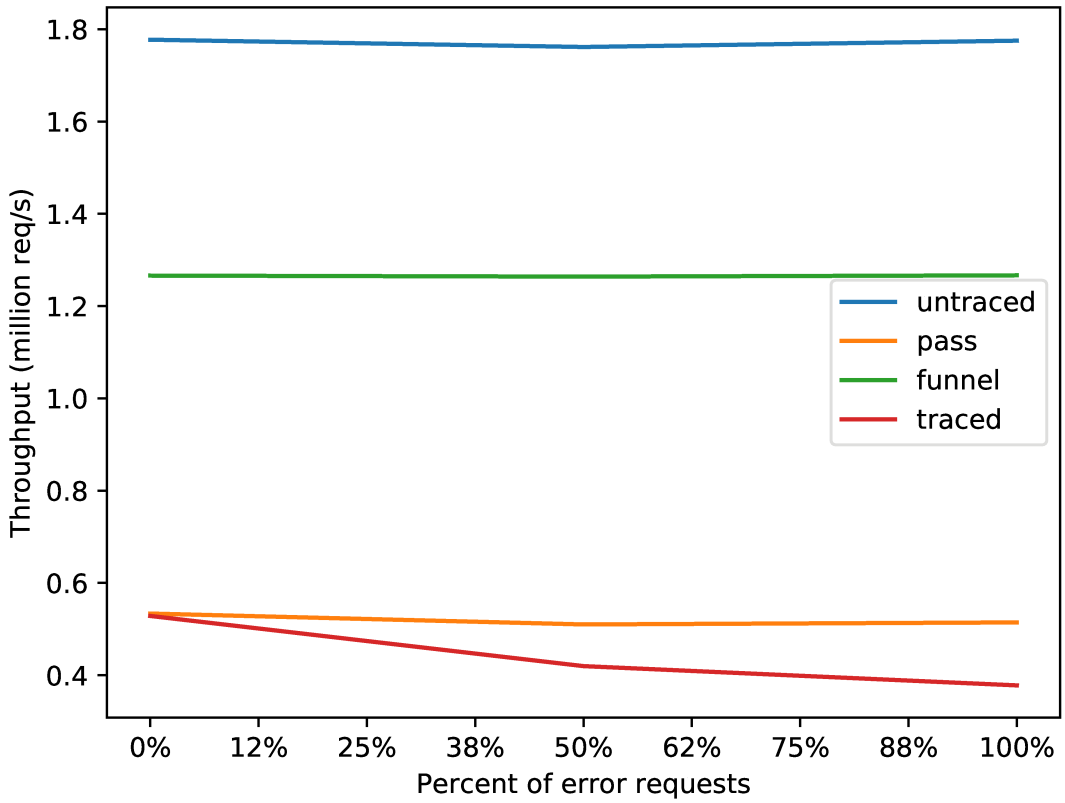

Figure 5.4: h2load HTTP throughput

The mean delay of benchmarked HTTP request is presented in the Figure 5.6. The effect of monitoring to the mean request delay is significant as well. The measured mean delay for monitored system is from 3.4x up to 4.7x higher than the reference. The delay increases along with the mirroring portion of data.

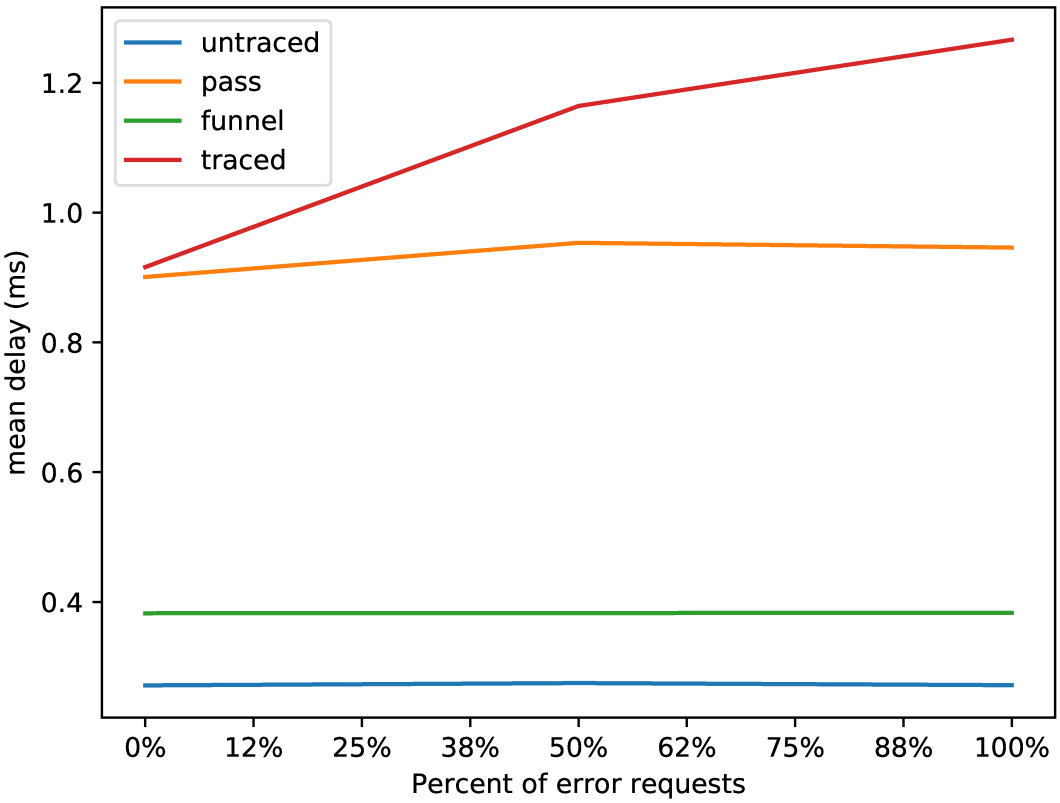

Figure 5.5: h2load mean request delay

The results for maximum delay are presented in the Figure 5.6. The maximum delay shows the same phenomena of occasional 200ms requests, which was observed on single host 99.999 percentile. The reference is at around 50ms while the monitored is slightly over 200ms. Unlike with the single host configuration, these outliers in delay do now show in the percentiles measured. The increase in maximum delay can be assumed to be result of the modifications in the traffic control as it was with the single host configuration.

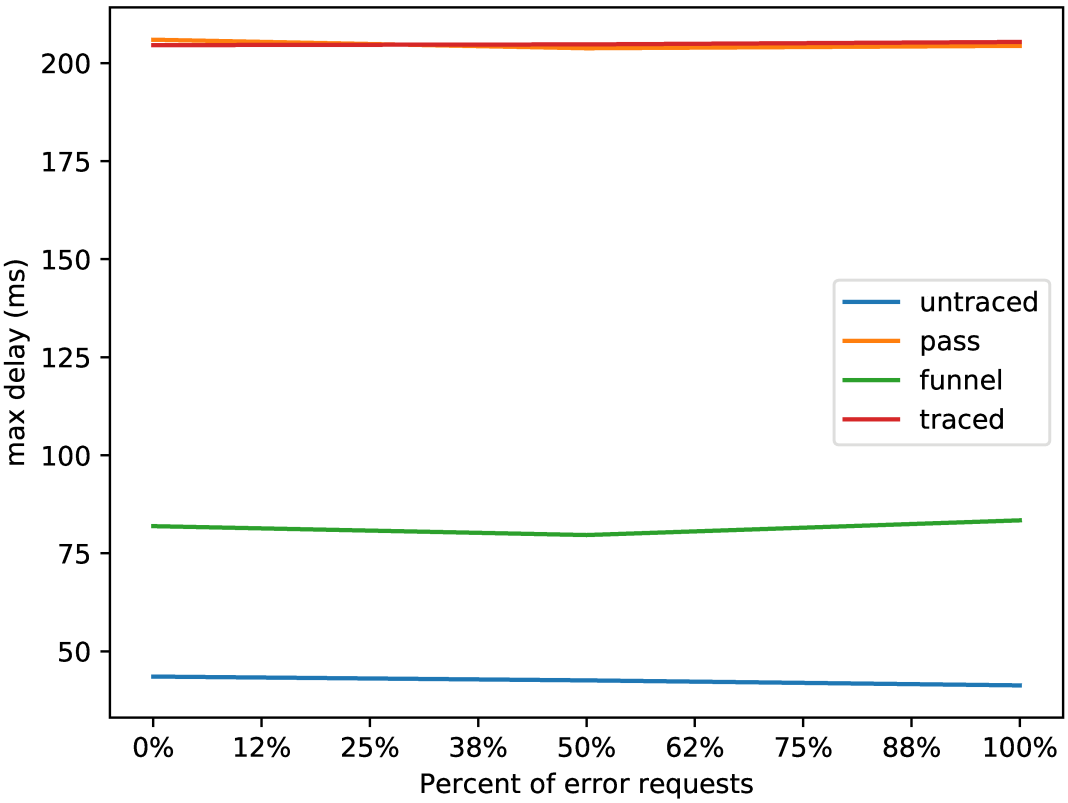

Figure 5.6: h2load max request delay

The distributions of request delays for the multi-host tests are presented in the Figure 5.7. The distributions show, that the shift towards higher latencies is significantly higher than, what was observed in the single host setup. Vast majority of the requests made without the monitor were complete in time span between 64 and 512 µs. With the monitor, the majority of requests were completed between 512 and 1024 µs.

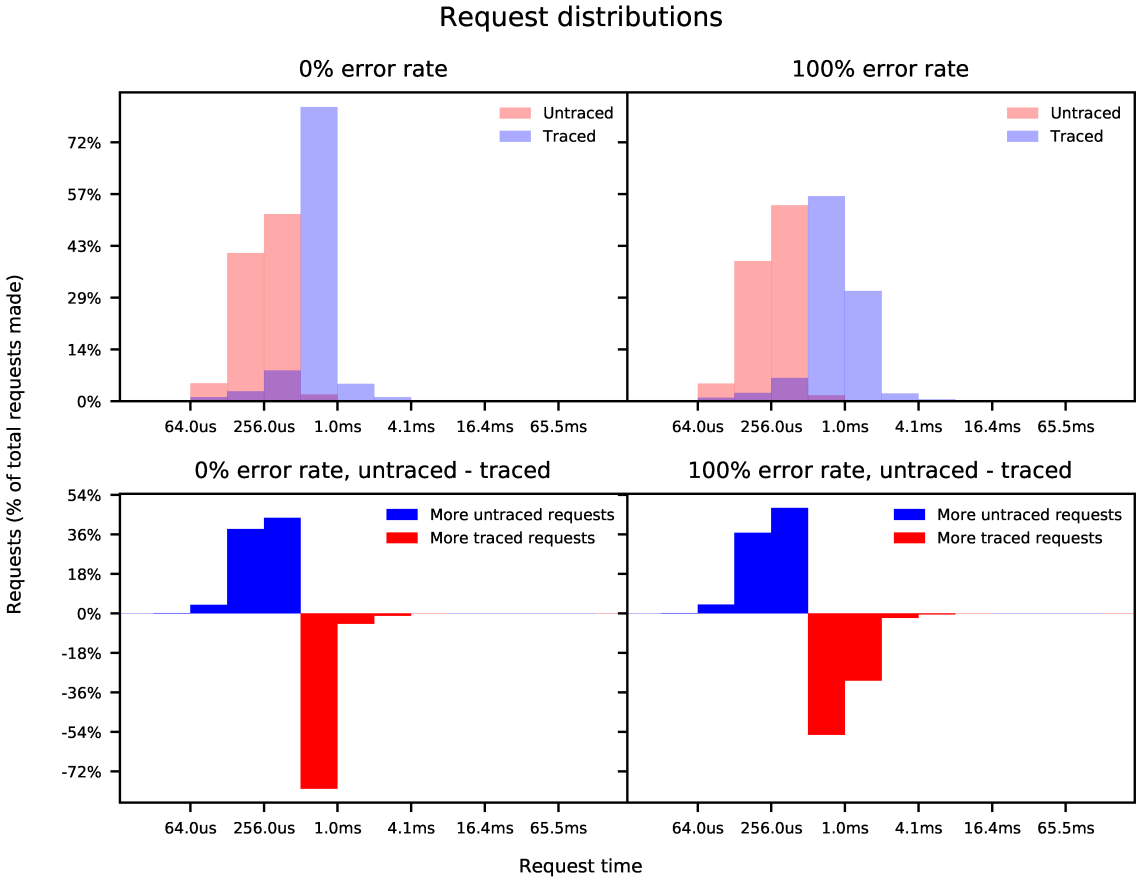

Figure 5.7: http request time distributions on log2 time scale. The second row shows the distribution difference between the monitored and non-monitored runs.

The percentiles for the request delays are presented in the tables 5.5 and 5.6. The median delay for the monitored http request is nearly 900ms. This is significant increase as the reference is under 270ms. Up to the 99.9 percentile, the delay for the non-monitored reference stays lower. However, past the 99.9 percentile, the delays for the monitored version are actually lower than for the reference. A probable cause for this is that the lower throughput of the monitored test results in less jitter to the delay. More research is however required for conclusive answers.

Percentile	50	90	99	99.9	99.99	99.999	99.9999
untraced	0.262	0.378	0.578	1	14	23	36
funnel	0.304	0.527	0.930	19	31	45	61
pass	0.877	1	2	4	5	6	13
traced	0.886	1	3	4	4	6	12

Table 5.5: HTTP request time (ms) percentiles with 0% errors

The with the overhead of mirroring is visible in the delays as well. The request delays for the 99.9 - 99.999 percentiles are greater for the mirrored requests than the monitored requests without mirroring. However, the additional overhead from mirroring is small when compared to the overhead of the monitoring itself.

Percentile	50	90	99	99.9	99.99	99.999	99.9999
untraced	0.266	0.374	0.562	0.977	15	23	36
funnel	0.300	0.530	0.971	18	30	45	61
pass	0.897	1	3	4	4	6	14
traced	1	2	4	7	9	10	15

Table 5.6: HTTP request time (ms) percentiles with 100% errors

5.3 Experimentation

Due to the large performance degradation caused by the eBPF monitor with the multi-host setup, additional experimentation was done identify the roots of the sudden increase in the overhead. The aim for these measurements were to identify the relationship of between the monitoring overhead and used CPUs.

The experimentation used exactly the same traffic loads and monitor, however the amount of CPU cores available to the traffic generators was varied. The results for these measurements are shown in the Figure 5.8 and 5.9. The results are a median from 30 independent runs of the same test.

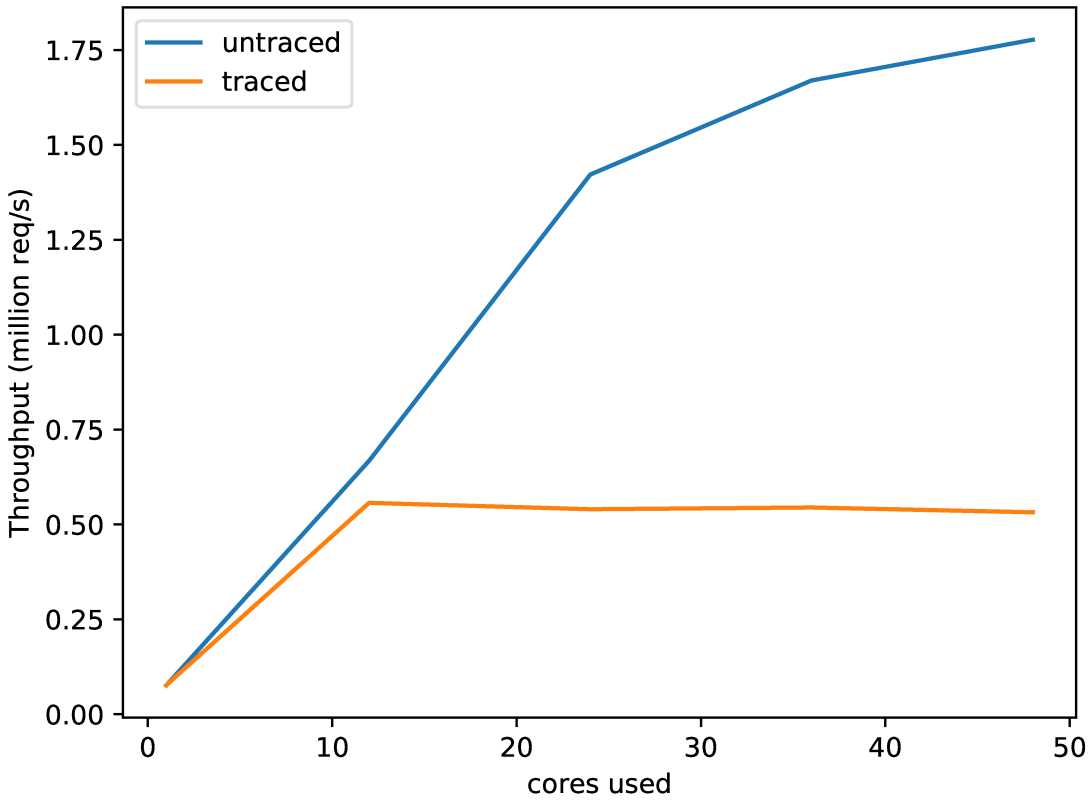

Figure 5.8: HTTP throughput as a function of used CPU cores

The measurements show that the non-monitored HTTP benchmark scales linearly up to the 24 cores. After 24 cores, the performance gain per core diminish. The monitored benchmark follows the non-monitored benchmark with only minor relative performance degradation to the 12 cores. After that there is no increase in throughput regardless of the number of cores used. while the increase in throughput stalls after 12 cores, the increase in delay certainly does not. In fact, the growth rate of mean delay seems to increase, as the gains in throughput stall.

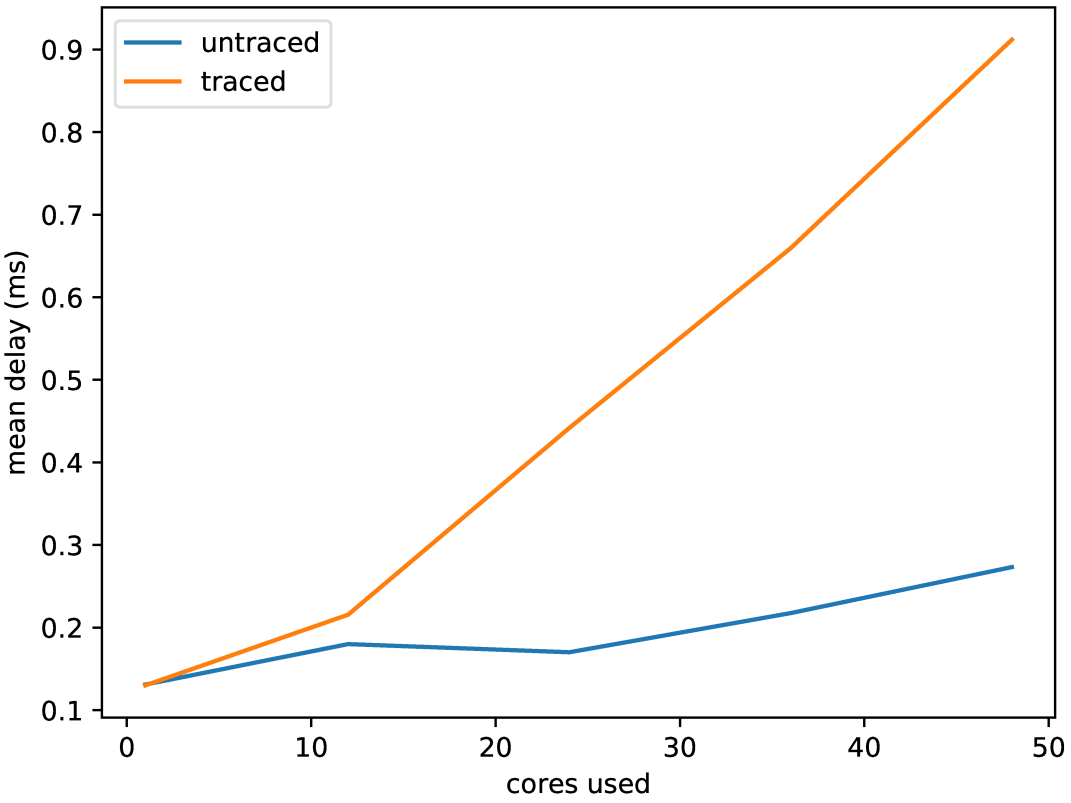

Figure 5.9: Mean delay of HTTP requests as a function of used CPU cores

5.3.1 Metrofunnel

The impact of the Metrofunnel monitor on the HTTP throughput, as shown in the Figure 5.4 is 30%. While the 30% drop in throughput is significant, it is much less than what was measured for the eBPF monitor. It seems that listening raw socket as done by Metrofunnel does not suffer from the same performance bottleneck as the traffic control based solution.

Chapter 6

Discussion

This chapter discusses the relationship of the Linux kernel based microservice monitoring and existing solutions for microservice monitoring. Furthermore, problems regarding the kernel based monitoring, their solutions and future tidings are discussed.

6.1 Performance

The results for performance overhead of the implemented eBPF monitor are mixed at best. The performance impact of at most 10% for throughput and negligible impact on delay are promising. However, the multi-host test setup introduced a grave performance degradation to the HTTP benchmark. The throughput was down to the 30% and the impact on delay significantly larger than with the single host setup. The characteristics of the results suggests that the eBPF monitor hit a hard performance bottleneck when 12 cores or more were in use. This effectively prevented any additional processors or HTTP client from positively contributing to the HTTP throughput. The non-monitored setup however did not suffer from this bottleneck.

Even though the underlying reason for the bottleneck was not identified, just the presence of the bottleneck has implications regarding the use case of a containerized monitor. An integral part of the use case is that the containerized monitor should work predictably and the same way regardless of the host environment. It is clear, that using traffic control and eBPF for monitoring can have unpredictable host specific side effects. These side effects could likely be mitigated by careful configuration of the host. This host specific configuration however is again in direct contradiction of the use case of a host independent monitoring container.

As a summary for the performance, the eBPF monitor has very low over-

head in the best-case scenario. However reliably achieving the best-case scenario in the context of a containerized monitor might not currently be possible without host specific configuration.

6.2 Complexity of the monitor

It has to be noted that the implemented monitor represented the simpler end of eBPF programs. It is likely that a production grade version would exceed this test implementation both in features and complexity. With the additional complexity for the monitor, it should be expected that the overhead will slightly increase as well. However, the production grade monitor is expected to be far further optimized than the monitor used in the test implementation. For example replacing the generic maps with per-cpu maps and IRQ-pinning should improve performance. Furthermore, the single host use case could have been optimized to only capture from either the ingress or egress instead of both, effectively halving the overhead.

6.3 Monitored interfaces

The problem of choosing the monitored interfaces in the hosts is all but trivial and depends on the container networking solution used. the problem is best illustrated using two containers residing in the same local docker network. In this case, there is three statically configured interfaces to which the monitor could be attached. First is the outgoing Ethernet interface. the second is the Docker0 bridge and the third is the bridge of the internal docker network. An eBPF monitor attached to any of these is unable see the traffic between the two containers. In the case of the internal docker network bridge this is unexpected. This is due to the fact that for example libpcap based solutions are able capture the traffic while traffic control filter cannot.

The end result of this is that effectively only place to get both internal and outgoing traffic is attaching the monitor to the virtual Ethernet interfaces. This solution is not without problems either. Firstly, an instance of the monitor has to be created every time a container is launched, while this can be done, it adds to the complexity of the monitoring solution. The other problem is that again, the local traffic which in optimal case would only be captured once, is necessarily captured on both ends. These specific complications cannot as such be generalized outside dockers implementation of the networking. There is however no reason to expect that the problem of choosing the captured network interfaces would be easier with other container

networking implementations.

6.4 Comparison to existing tools

6.4.1 Network Monitoring

This section compares the eBPF based solution to a hypothetical implementation constructed using older and more tested tools. To be able to reasonably compare the implementations, a set of feasibility criteria is needed. The most important criteria is arguably the performance impact. Other criteria to consider are flexibility, maintainability and quality. In the scope of the comparison, flexibility is defined as the ability of a single solution to be extended into multiple monitoring related use cases. Maintainability on the other hand is in this scope used to refer how easy the solution is to implement or change in general. Quality refers here to the extent that the solution fulfills its use case.

As a rule of thumb, most of the use cases for the eBPF could be implemented with more tested tools or with a combination of them. However usually a compromise regarding one or more of aforementioned criteria is required. For example, the selective network mirroring based on the HTTP error detection could have been implemented with libpcap or the standard tools using libpcap such as tcpdump or wireshark.

Let us first consider an implementation where the performance of the libpcap solution is required to match the eBPF monitor. It is possible to construct libpcap filter matching the HTTP headers in a similar fashion as the eBPF program. This is due to the fact that the filters of libpcap use cBPF which is the older and more limited variant of eBPF, which is now implicitly converted to eBPF by the Linux Kernel. Due to the limitations of cBPF, the filter is however not able to use maps to maintain state. This means that the list of mirrored connections needs to be maintained by user space program instead of the kernel. The mirroring cannot be done with the in-kernel cBPF filter installed by libpcap either, as the filter may only decide whether to forward a packet to user space or not. Mirroring in this use case could be handled by manually wrapping the data in user space and sending it forward. An alternative would be to manage the mirrored connections by adding a mirroring rule to iptables, or traffic control from user space for each mirrored connection.

The major quality issue with the manual wrapping of data obtained from libpcap is the stateless nature of cBPF filter. To obtain the packets following the error the filter has to be modified to include the rules for the mirrored

connection or a new filter must be added. At this point all of the presented solutions require adding some configuration to the kernel either get the data to user space or to directly mirror it somewhere. The problem itself is that adding the rules to the kernel is slow regardless of whether libpcap, traffic control or iptables is configured from user space. It is very likely that most of the relevant follow up for the error has already passed before the new configuration is in effect, which is a huge downgrade in quality when compared to the eBPF filter. In conclusion, any libpcap based solution aiming to compete with performance of the eBPF will have to resort to manipulating the kernel tool configurations which will result in a significant quality degradation. In addition to the quality, the maintainability and flexibility of the solutions described above cannot really compete with the eBPF based solution.

The only conceivable way or matching the quality and the maintainability of the best case of the eBPF monitor would be to just capture all TCP traffic with libpcap and then do the filtering and mirroring of the data in user space. Most of the existing libpcap based implementations indeed seem to resort to this approach using at most very coarse static filters and doing the rest of the processing in user space. [42][4][17][38][11]

Due to the overhead of copying packets to the user space which the performance overhead of should be greater than measured for the eBPF monitor. The results obtained from the single host benchmarks indeed support this, as the difference in HTTP throughput was a maximum of 10% overhead against the 50% for the Metrofunnel. The multi-host benchmark however demonstrated that there are configurations on which the performance of the eBPF monitor is so far from optimal, that a libpcap based solution, such as Metrofunnel [28], may easily outperform it. Overhead larger than 70% is highly unusual when compared to the existing literature. Even monitoring packet data with eBPF by tcpsend function with a Kprobe, which should be much more costly, has been done for large packets with only 50% performance overhead [13].

6.4.2 Microservice monitoring

Most of the tools currently used to monitor microservices do not use the network monitoring approach, but instead rely instrumenting the application or indirect data collection methods for the monitoring.

It is hard for the eBPF monitor to compete with existing tools at the performance overhead or the quality of data. The monitoring tools instrumenting the applications have access to the full context of any error occurring whereas, the eBPF monitor must keep track of any context it requires to function. Furthermore, for the eBPF monitor in networking context, the

base of cost is measured per packet, whereas by application instrumentation the cost is at most per request. The lower overhead combined with easier access to context ensure, that it is hard to advocate using eBPF monitor in cases where components used are supported by the already production grade tools. To further show the immaturity of eBPF, recently there was discovered a bug where eBPF would give wrong result on something as basic as subtract operation [19].

The eBPF monitor is not able nor meant to be a replacement for the existing solutions. Integrating parts of the monitor as data sources to existing tools could provide useful metrics without the cost of building a whole new monitoring solution. An example solution for this would be outputting statistics aggregated via eBPF program as metrics to Prometheus.

Another possible target for integration is Cilium [10]. Cilium is an application firewall and overlay network, which is implemented on top of eBPF. The eBPF based implementation could yield synergy between the implementations for example with shared eBPF maps. This is especially true for solutions already utilizing Cilium for example as the overlay network for a Kubernetes implementation.

6.5 Containerization and Security implications

From the dependency perspective the monitoring implementation was successful. The only unavoidable dependency to the host system of the monitor is sufficiently new kernel version to support the eBPF used. Kernel sources and headers required to run eBPF can to some extent be shipped with the container image. In the end, the capabilities of the host system determine the reasonable approach considering the kernel sources. Including the kernel headers in the for all used kernel versions in the monitors container image can definitely be done as long as the used versions are well known. This however complicates the image usage as keeping the headers up to date with the used kernel versions is a non-trivial task. In any use case where the sources can be mounted to the image from the host system, it is likely the better option. Despite being more attractive option than the alternatives, mounting the sources from the host is not perfect solution either. At least Fedora 29 removes the older sources while updating kernel version. The running kernel on the other hand, is only updated at the next system restart. This leaves a period where necessary sources for eBPF program compilation are not found in the system.

From the security point of view, the isolation of the monitor was not very successful. Even the networking context required SYS_ADMIN and

NET_ADMIN to be given to the monitor container. The SYS_ADMIN capability alone indicates that the container should be in practice considered as running with full root access. If other hook points for eBPF such as kprobes or uprobes are to be used, additional debugfs mounts and possibly access to the hosts process namespace are required. For example, in a use case where an Uprobe needs to be inserted in to an executable inside another container, access to the root file system of the monitored process is required.

It is important to highlight that tweaking security profiles with SELinux [53] or AppArmor [2] was not included in the scope of this thesis. These tools are important part of securing privileged containers [25]. While these tools can be used for implementing incredibly fine-grained policies and rules, using them to secure the monitor is hard. In the best case the ruleset for the access control requires constant maintenance and even that might not be enough. This is because any capability required for the monitor to function might at the same time compromise the container sandbox. The eBPF itself is a good example of this. Even the ability just to run arbitrary eBPF programs can be enough to compromise the system. One such way is a consequence of the fact that many tracing related bpf programs may write to arbitrary user space memory addresses using user_probe_write function. This has been shown to enable privilege escalation attacks [14].

6.6 Future research

6.6.1 eBPF offloading

eBPF offloading refers to outsourcing the execution of the eBPF programs away from the CPU. Offloading the traffic control decision making and the related eBPF programs to the NIC could allow for drastically reduced impact for processing overhead. The support for transparently offloading TC eBPF programs is already present in the Linux kernel. However currently only the smart NICs produced by Netronome support the offloading [35].

Considering the monitor proposed in this thesis, the offloading could help to scale the monitor for higher speeds and mitigate some of the issues encountered with the highly parallel traffic loads. Due to the lacking support and poor maturity, the offloading is not yet viable approach to be considered.

6.6.2 Alternative triggers and actions

The triggers and actions implemented to the monitor represented a minor subset of possible interactions. The only implemented action was the traffic

mirroring and the only trigger was error codes found in HTTP requests. The connection between the trigger and action was an eBPF map keeping track of which connections to mirror. The error requests or even network monitoring are however by no means only possible triggers. An eBPF map may be shared with other eBPF programs with different contexts or user space programs.

In the networking context, alternative trigger could be a probe triggering at certain probability at new TCP connections, effectively working as a TCP connection sampler. Outside the networking context, the instrumentation with Kernel or user space probes could provide possible triggers. Without access to the networking context however the triggering events must be possible to correlate with the corresponding network traffic.

Chapter 7

Conclusions

In this thesis, monitoring 5G microservices with Linux kernel was researched through the use case of an eBPF based network monitor. Both the tooling for microservice monitoring and Linux kernel are actively improved independent of each other.

The eBPF benchmarks show that from performance point of view, the eBPF has potential to allow flexible monitoring with negligible performance impact. However, utilizing eBPF reliably to that extent proved to be a non-trivial task. The benchmark results for multi host setup showed HTTP throughput decrease below 30% due to the presence of an eBPF monitor. This was shown to be linked to the number of cores used for the benchmark, while the exact reason could not be identified. However, this significant inconsistency in the overhead, which depends on the used hardware and configuration, confirms that unless an effort is made to configure the host server and the monitor for optimal settings, the lowest overheads will not be achieved consistently.

One of the main reasons for running services in containers is the promise of similar behavior regardless of the host environment. The inconsistency in performance, depending on the capabilities of the host server, makes the eBPF monitor less than ideal use case for containerized application.

When compared to the existing tools for microservice monitoring, the advantages provided by eBPF monitor are at best questionable. The low-level monitoring is problematic due to the fact that the context of events tend to be harder to access. The existing tools most often resort to instrumenting the service applications or indirect monitoring, such as log analysis. Instrumenting the application has the advantage of of having the full context of the service available for the monitor. Furthermore, the performance overhead is low and consistent between the environments, as the application or instrumentation framework is responsible the monitoring.

The development of the eBPF in the Linux kernel is not complete. New features are constantly introduced and old ones improved. Furthermore, many of existing tools, which from performance point of view would benefit from eBPF, is not yet adapted to use it. The awareness of containers in the operating system level monitoring tools is in a similar situation. As the awareness for containers and utilization of eBPF for existing tools improves, it is to be expected that kernel level monitoring of microservices will become more viable option than it is now.

Bibliography

[1] Agiwal, M., Roy, A., and Saxena, N. Next generation 5g wireless networks: A comprehensive survey. *IEEE Communications Surveys & Tutorials 18*, 3 (2016), 1617–1655. DOI: https://doi.org/10.1109/COMST.2016.2532458.

[2] Apparmor repository. https://gitlab.com/apparmor. Accessed: 2019-02-28.

[3] Blunck, J., Desnoyers, M., and Fournier, P.-M. Userspace application tracing with markers and tracepoints. In *Proceedings of the Linux Kongress* (2009), pp. 7–14.

[4] Bonelli, N., Giordano, S., and Procissi, G. Enabling packet fan-out in the libpcap library for parallel traffic processing. In *2017 Network Traffic Measurement and Analysis Conference (TMA)* (2017), IEEE, pp. 1–9. DOI: https://doi.org/10.23919/TMA.2017.8002904.

[5] Bcc project, github. https://github.com/iovisor/bcc. Accessed: 2019-04-25.

[6] Brown, M. A. Traffic control howto. *Guide to IP Layer Network* (2006), 49.

[7] Capabilities manual page. https://man7.org/linux/man-pages/man7/capabilities.7.html. Accessed: 2018-11-29.

[8] Cgroup manual page. https://man7.org/linux/man-pages/man7/cgroups.7.html. Accessed: 2019-04-30.

[9] Bpf and xdp reference guide. https://cilium.readthedocs.io/en/latest/bpf/. Accessed: 2018-10-24.

[10] Cilium home. https://cilium.io/. Accessed: 2019-04-12.

[11] Cinque, M., Della Corte, R., Iorio, R., and Pecchia, A. An exploratory study on zeroconf monitoring of microservices systems. In *2018 14th European Dependable Computing Conference (EDCC)* (2018), IEEE, pp. 112–115. DOI: https://doi.org/10.1109/EDCC.2018.00028.

[12] Claassen, J., Koning, R., and Grosso, P. Linux containers networking: Performance and scalability of kernel modules. In *NOMS 2016-2016 IEEE/IFIP Network Operations and Management Symposium* (2016), IEEE, pp. 713–717. DOI: https://doi.org/10.1109/NOMS.2016.7502883.

[13] Clément, B. Linux kernel packet transmission performance in high-speed networks, 2016.

[14] Dileo, J., and Andy, O. Kernel tracing with ebpf. https://fahrplan.events.ccc.de/congress/2018/Fahrplan/system/event_attachments/attachments/000/003/729/original/Kernel_Tracing_With_eBPF.pdf. Accessed: 2019-02-25.

[15] Docker swarm. https://docs.docker.com/engine/swarm/. Accessed: 2019-04-26.

[16] Dronamraju, S. Uprobe-tracer: Uprobe-based event tracing. https://www.kernel.org/doc/Documentation/trace/uprobetracer.txt. Accessed: 2019-04-23.

[17] Duarte, V., and Farruca, N. Using libpcap for monitoring distributed applications. In *2010 International Conference on High Performance Computing & Simulation* (2010), IEEE, pp. 92–97. DOI: https://doi.org/10.1109/HPCS.2010.5547144.

[18] ebpf by brendan gregg. http://www.brendangregg.com/eBPF/linux_ebpf_internals.png. Accessed: 2019-03-01.

[19] ebpf subtract bug. https://lore.kernel.org/netdev/CAJPywTJqP34cK2OiLM5YmUMz9KXQOdu1-+BZrGMAGgLuBWz7fg@mail.gmail.com/. Accessed: 2019-05-15.

[20] Elk home page. https://www.elastic.co/elk-stack. Accessed: 2019-04-10.

[21] Erlacher, F., Woertz, S., and Dressler, F. A tls interception proxy with real-time libpcap export. In *41st IEEE Conference on Local Computer Networks (LCN 2016), Demo Session* (2016).

[22] Flannel repository. `https://github.com/coreos/flannel`. Accessed: 2019-03-04.

[23] Gebai, M., and Dagenais, M. R. Survey and analysis of kernel and userspace tracers on linux: Design, implementation, and overhead. *ACM Computing Surveys (CSUR) 51*, 2 (2018), 26. DOI: `https://doi.org/10.1145/3158644`.

[24] Hausenblas, M. *Container Networking*. O'Reilly Media, 2018.

[25] Hertz, J. Abusing privileged and unprivileged linux containers. *Whitepaper, NCC Group 48* (2016).

[26] Hu, Y. C., Patel, M., Sabella, D., Sprecher, N., and Young, V. Mobile edge computing—a key technology towards 5g. *ETSI white paper 11*, 11 (2015), 1–16.

[27] Hubert, B. Traffic control, manual page. `https://linux.die.net/man/8/tc`. Accessed: 2019-02-26.

[28] Iorio, R. Real-time monitoring of microservices-based software systems. Master's thesis, University of Naples Federico II, 2017.

[29] Iperf3 at dockerhub. `https://hub.docker.com/r/networkstatic/iperf3/`. Accessed: 2018-12-07.

[30] Iperf home page. `https://iperf.fr/`. Accessed: 2018-11-28.

[31] Iperf3 manual page. `https://www.mankier.com/1/iperf3`. Accessed: 2018-12-07.

[32] Jaeger home page. `https://www.jaegertracing.io`. Accessed: 2019-03-01.

[33] Keniston, J., Panchamukhi, P. S., and Hiramatsu, M. Kernel probes (kprobes). `https://github.com/torvalds/linux/blob/v4.18/Documentation/kprobes.txt`. Accessed: 2018-10-24.

[34] Khan, A. Key characteristics of a container orchestration platform to enable a modern application. *IEEE Cloud Computing 4*, 5 (2017), 42–48. DOI: `https://doi.org/10.1109/MCC.2017.4250933`.

[35] Kicinski, J., and Viljoen, N. ebpf hardware offload to smartnics: cls bpf and xdp. *Proceedings of netdev 1* (2016).

[36] Kubernetes documentation, cluster networking. `https://kubernetes.io`. Accessed: 2019-04-26.

[37] Kubernetes documentation, cluster networking. `https://kubernetes.io/docs/concepts/cluster-administration/networking/`. Accessed: 2019-03-04.

[38] Laboshin, L., Lukashin, A., and Zaborovsky, V. The big data approach to collecting and analyzing traffic data in large scale networks. *Procedia Computer Science 103* (2017), 536–542. DOI: `https://doi.org./10.1016/j.procs.2017.01.048`.

[39] Linux namespaces, manual page. `http://man7.org/linux/man-pages/man7/namespaces.7.html`. Accessed: 2019-03-05.

[40] Lttng home page. `https://lttng.org/`. Accessed: 2019-04-26.

[41] Mount namespace, manual page. `http://man7.org/linux/man-pages/man7/mount_namespaces.7.html`. Accessed: 2019-03-05.

[42] Naik, P., Shaw, D. K., and Vutukuru, M. Nfvperf: Online performance monitoring and bottleneck detection for nfv. In *2016 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)* (2016), IEEE, pp. 154–160. DOI: `https://doi.org/10.1109/NFV-SDN.2016.7919491`.

[43] Network namespace, manual page. `http://man7.org/linux/man-pages/man7/network_namespaces.7.html`. Accessed: 2019-03-04.

[44] Openvswitch homepage. `https://www.openvswitch.org/`. Accessed: 2019-03-04.

[45] Kernel packet classifier header. `https://github.com/torvalds/linux/blob/v4.18/include/uapi/linux/pkt_cls.h`. Accessed: 2018-11-29.

[46] Pid namespace, manual page. `http://man7.org/linux/man-pages/man7/pid_namespaces.7.html`. Accessed: 2019-03-05.

[47] Pid namespace, manual page. `http://man7.org/linux/man-pages/man7/user_namespaces.7.html`. Accessed: 2019-03-05.

[48] Prometheus home page. `https://prometheus.io/`. Accessed: 2019-04-10.

[49] Pyroute2 documentation. `https://docs.pyroute2.org/`. Accessed: 2019-04-17.

[50] Rotter, C., Farkas, L., Nyíri, G., Csatári, G., Jánosi, L., and Springer, R. Using linux containers in telecom applications. In *Proc. ICIN* (2016), pp. 234–241.

[51] Satyanarayanan, M., Bahl, P., Caceres, R., and Davies, N. The case for vm-based cloudlets in mobile computing. *IEEE pervasive Computing*, 4 (2009), 14–23. DOI: `https://doi.ieeecomputersociety.org/10.1109/MPRV.2009.82`.

[52] Seccomp manual page. `https://man7.org/linux/man-pages/man2/seccomp.2.html`. Accessed: 2019-04-30.

[53] Selinux repository. `https://github.com/SELinuxProject`. Accessed: 2019-02-28.

[54] Simsek, M., Aijaz, A., Dohler, M., Sachs, J., and Fettweis, G. 5g-enabled tactile internet. *IEEE Journal on Selected Areas in Communications 34*, 3 (2016), 460–473. DOI: `https://doi.org/10.1109/JSAC.2016.2525398`.

[55] Systemtap home page. `https://sourceware.org/systemtap/`. Accessed: 2019-04-26.

[56] Taleb, T., Samdanis, K., Mada, B., Flinck, H., Dutta, S., and Sabella, D. On multi-access edge computing: A survey of the emerging 5g network edge cloud architecture and orchestration. *IEEE Communications Surveys & Tutorials 19*, 3 (2017), 1657–1681. DOI: `https://doi.org/10.1109/COMST.2017.2705720`.

[57] Tcpdump and libpcap home page. `https://www.tcpdump.org/`. Accessed: 2019-04-24.

[58] Toupin, D. Using tracing to diagnose or monitor systems. *IEEE software 28*, 1 (2011), 87–91. DOI: `https://doi.org/10.1109/MS.2011.20`.

[59] Zhang, J., and Moore, A. Traffic trace artifacts due to monitoring via port mirroring. In *2007 Workshop on End-to-End Monitoring Techniques and Services* (2007), IEEE, pp. 1–8. DOI: `https://doi.org/10.1109/E2EMON.2007.375317`.

[60] Zhao, J., Yin, C., Jin, X., and Liu, W. A high-speed network data acquisition system based on big data platform. In *2017 IEEE International Conference on Computational Science and Engineering (CSE) and IEEE International Conference on Embedded and Ubiquitous Computing (EUC)* (2017), vol. 1, IEEE, pp. 677–681. DOI: `https://doi.org/10.1109/CSE-EUC.2017.128`.

[61] Zhao, Y., Xia, N., Tian, C., Li, B., Tang, Y., Wang, Y., Zhang, G., Li, R., and Liu, A. X. Performance of container networking technologies. In *Proceedings of the Workshop on Hot Topics in Container Networking and Networked Systems* (2017), ACM, pp. 1–6.

[62] Zipkin home page. `https://zipkin.io`. Accessed: 2019-03-01.